O'REILLY®

3-е издание

MySQL

ПО МАКСИМУМУ

ОПТИМИЗАЦИЯ, РЕПЛИКАЦИЯ, РЕЗЕРВНОЕ КОПИРОВАНИЕ

Бэрон Шварц
Петр Зайцев
Вадим Ткаченко

ПИТЕР®

THIRD EDITION

High Performance MySQL

Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

MySQL

ПО МАКСИМУМУ

ОПТИМИЗАЦИЯ, РЕПЛИКАЦИЯ, РЕЗЕРВНОЕ КОПИРОВАНИЕ

3-е издание

Бэрон Шварц
Петр Зайцев
Вадим Ткаченко

ПИТЕР Санкт-Петербург · Москва · Екатеринбург · Воронеж
Нижний Новгород · Ростов-на-Дону · Самара · Минск

2018

Бэрон Шварц, Петр Зайцев, Вадим Ткаченко

MySQL по максимуму

3-е издание

Серия «Бестселлеры O'Reilly»

Перевел с английского *А. Колышкин*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художественный редактор	*С. Заматевская*
Корректор	*Е. Павлович*
Верстка	*Г. Блинов*

ББК 32.988-02-018 УДК 004.738.5

Шварц Б., Зайцев П., Ткаченко В.

Ш33 MySQL по максимуму. 3-е изд. — СПб.: Питер, 2018. — 864 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-0696-7

Хотите выжать из MySQL максимум возможностей?

Вам поможет уникальная книга, написанная экспертами для экспертов.

Познакомьтесь с продвинутыми приемами работы с MySQL: разработкой схем, индексов и запросов для настройки сервера, операционной системы и аппаратной части, способами масштабирования приложений и репликацией, балансировкой нагрузки, обеспечением доступности и восстановлением после отказов.

Прочитав эту книгу, вы узнаете, почему MySQL устроена именно так, познакомитесь с разбором практичных кейсов, научитесь мыслить на одном языке с вашей базой данных.

Бестселлер Шварца, Зайцева и Ткаченко — книга, необходимая любому профессионалу и способная превратить самую страшную «нештатную ситуацию» в легко преодолимый «рабочий момент».

Читайте и совершенствуйтесь!

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1449314286 англ.

ISBN 978-5-4461-0696-7

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 191123, Россия, г. Санкт-Петербург, ул. Радищева, д. 39, к. Д, офис 415. Тел.: +78127037373.

Дата изготовления: 04.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», РБ, 220020, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 29.03.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 69,660. Тираж 1000. Заказ 2880.

Отпечатано в АО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор». 142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8(499)270-73-59

Краткое содержание

Приложения

Оглавление

Приложения

Предисловие

Я уже много лет являюсь поклонником этой отличной книги, а к третьему изданию она стала еще лучше. Эксперты мирового уровня не только поделились своим опытом, но и нашли время для обновления книги. В ней дается множество рекомендаций относительно того, как добиться от MySQL высокой производительности, но основное внимание уделяется процессу оптимизации, а не фактам и мелочам. Эта книга поможет вам понять, как улучшить работу независимо от изменений в поведении MySQL.

Такой замечательной книга получилась благодаря уникальной квалификации авторов. В издании соединились их богатый опыт, принципиальный подход, ориентация на эффективность и стремление к самосовершенствованию. Говоря об *опыте*, я имею в виду то, что авторы работали над улучшением производительности MySQL с тех самых пор, когда она еще не масштабировалась, до текущего момента, когда ситуация существенно изменилась к лучшему. Под *принципиальным подходом* я понимаю тот факт, что они рассматривают эту деятельность как научную, то есть сначала определяют проблемы, которые необходимо решить, а затем решают их, проводя соответствующие исследования.

Больше всего меня поражает стремление авторов к *эффективности*. Будучи консультантами, они ценят каждую минуту своего времени. Клиенты, оплачивающие каждый час работы, хотят, чтобы их проблемы решались быстро. В итоге авторы вывели свою технологию и создали инструменты для качественной и эффективной работы. В данной книге описана эта технология и приведен исходный код инструментов.

Наконец, они продолжают совершенствовать свои навыки. Это выражается в смещении акцентов с пропускной способности на время отклика, изучении производительности MySQL на новом оборудовании и получении новых знаний, в частности, по теории массового обслуживания, которая может применяться для оценки производительности.

Я полагаю, что эта книга — предвестник светлого будущего для MySQL. Поскольку MySQL развивается, чтобы быть способной поддерживать увеличивающиеся рабочие нагрузки, авторы книги приложили усилия к тому, чтобы сообщество разработчиков MySQL стало лучше понимать, как повысить производительность этой системы. Они даже внесли в это свой вклад, разработав продукты XtraDB и XtraBackup. Я продолжаю учиться у них и надеюсь, что вы тоже не пожалеете времени на это.

Марк Каллаган (Mark Callaghan),
инженер-программист, Facebook

Введение

Написанная нами книга ориентирована на потребности создателей приложений MySQL. При этом мы учитывали требования администраторов баз данных. Мы рассчитываем, что вы уже работали с MySQL и имеете некоторый опыт системного администрирования, работы с сетями и операционными системами семейства UNIX.

Второе издание содержало большое количество информации, но ни одна книга не может полностью охватить рассматриваемую тему. В промежутке между вторым и третьим изданиями мы сделали множество заметок по тысячам интересных задач, которые решали и мы, и наши знакомые. Когда же начали делать наброски третьего издания, поняли, что для изложения всего материала потребуется от 3000 до 5000 страниц, но при этом книга все равно не будет полной. Поразмышляв над проблемой, мы пришли к выводу, что в стремлении во втором издании к глубокому изложению тем мы фактически ограничили себя в том смысле, что не часто объясняли, *как все это применять на практике*.

В результате третье издание отличается от второго. Мы по-прежнему даем много материала и все так же придаем большое значение надежности и безошибочности. Однако, помимо этого, мы попытались внести в книгу еще более глубокую идею: хотим объяснить не просто как работает MySQL, но и почему она так работает. Мы включили в издание больше разнообразных историй и кейсов, которые демонстрируют принципы MySQL. Основываясь на этих принципах, мы попытались ответить на такие вопросы: каких результатов можно достичь при реальном использовании MySQL с заданной внутренней архитектурой, почему эти результаты имеют значение, становится ли при этом MySQL более (или менее) подходящей для решения конкретных задач.

В конечном счете мы надеемся, что знание внутреннего устройства MySQL поможет вам в ситуациях, не описанных в этой книге. Кроме того, мы рассчитываем, что новое понимание MySQL поможет в теории и на практике освоить методы проектирования и поддержки систем, а также научиться находить и устранять проблемы в них.

Структура книги

В этой книге освещено множество сложных тем. Сейчас объясним, как мы упорядочили их для упрощения работы.

Общий обзор

Глава 1, «История и архитектура MySQL», посвящена основам, которые необходимо усвоить, прежде чем приступать к более сложным темам. Для того чтобы эффективно использовать MySQL, вы должны понимать, как она устроена. В этой главе рассматриваются архитектура MySQL и ключевые особенности ее подсистем хранения. Приводятся базовые сведения о реляционных базах данных, а также

о транзакциях. Эта глава также может выступать в роли введения в MySQL, если вы уже знакомы с какой-нибудь другой СУРБД, например Oracle. Кроме того, в этой главе мы обозначили исторический контекст: изменения в MySQL с течением времени, недавнюю смену владельцев и то, что, по нашему мнению, произойдет с ней в дальнейшем.

Закладка фундамента

В следующих главах приведен материал, к которому вы будете обращаться снова и снова в процессе использования MySQL.

В главе 2, «Эталонное тестирование MySQL», даются базовые сведения об эталонном тестировании. Здесь описывается методика определения того, какого рода нагрузки способен выдерживать сервер, насколько быстро он может выполнять конкретные задачи и т. п. Умение выполнять эталонное тестирование — это важный навык, который помогает оценивать то, как сервер работает под нагрузкой. Но не менее важно знать, когда оно будет бесполезным.

Глава 3, «Профилирование производительности сервера», знакомит вас с определением времени отклика — подходом, который применяется для выявления неполадок и диагностики проблем, связанных с производительностью сервера. Важность этого подхода доказана при решении нескольких наиболее загадочных проблем из тех, с которыми мы сталкивались. Возможно, вы захотите модифицировать наш подход (в конце концов, мы разработали его, в свою очередь изменив подход Кэри Миллсапа (Cary Millsap)).

В главах 4–6 мы рассмотрим три темы, которые вместе создают основу для хорошего логического и физического проектирования баз данных. В главе 4, «Оптимизация схемы и типов данных», описаны различные нюансы типов данных, проектирования таблиц и индексов. В главе 5, «Повышение производительности с помощью индексирования», речь пойдет об индексах, то есть о проектировании физической базы данных. Четкое понимание того, что такое индексы и как их применять, очень важно для эффективного использования MySQL, поэтому вы, вероятно, впоследствии захотите вернуться к этой главе. В главе 6, «Оптимизация производительности запросов», речь пойдет о том, как MySQL выполняет запросы и как можно воспользоваться сильными сторонами оптимизатора запросов. В этой главе приведено много конкретных примеров почти всех типичных запросов, иллюстрирующих оптимальную работу MySQL и показывающих, как преобразовать запросы в такую форму, чтобы задействовать максимум возможностей СУБД.

Все упомянутые нами до этого момента темы — таблицы, индексы, данные и запросы — могут относиться к любым системам управления базами данных. В главе 7, «Дополнительные возможности MySQL», мы выйдем за пределы основ и покажем, как использовать расширенные возможности MySQL. Мы рассмотрим кэш запросов, хранимые процедуры, триггеры, кодировки и пр. Эти средства реализованы в MySQL иначе, чем в других СУРБД, и хорошее их понимание откроет перед вами новые возможности повышения производительности, о которых вы, быть может, даже не задумывались.

Настройка приложения

В следующих двух главах обсуждается, как заставить ваше приложение хорошо работать на вашем оборудовании. В главе 8, «Оптимизация параметров сервера», мы обсудим, как настроить MySQL, чтобы извлечь максимум возможного из имеющейся аппаратной конфигурации сервера в применении к конкретному приложению. В главе 9, «Оптимизация операционной системы и оборудования», объясняется, как выжать все, что только можно, из операционной системы и используемого вами оборудования. Мы подробно обсудим твердотельные хранилища и предложим аппаратные конфигурации, которые могут обеспечить наилучшую производительность для больших приложений.

В обеих этих главах изучаются определенные особенности внутреннего устройства MySQL. Мы неоднократно будем возвращаться к данной теме, в том числе в приложениях.

MySQL как компонент инфраструктуры

MySQL существует не в вакууме, это часть общего стека приложений. Для своего приложения вам нужно создать надежную общую архитектуру. Следующие несколько глав посвящены тому, как это сделать.

В главе 10, «Репликация», мы обсудим убойную возможность MySQL — способность настроить несколько серверов так, чтобы они были синхронизированы с главным сервером. К сожалению, репликация — это, пожалуй, самая неприятная для некоторых функция MySQL. Однако это не всегда сложная тема, и мы покажем, как обеспечить хорошую работу репликации.

В главе 11, «Масштабирование MySQL», рассказывается, что такое масштабируемость (это не то же самое, что производительность), почему приложения и системы не масштабируются и как с этим бороться. Если вы все сделаете правильно, то сможете масштабировать MySQL так, чтобы реально было достичь практически любых целей. В главе 12, «Высокая доступность», раскрывается связанная, но все-таки другая тема, а именно: как обеспечить, чтобы MySQL оставалась активной и правильно функционировала. Из главы 13, «MySQL в облаке», вы узнаете, что изменяется при запуске MySQL в средах облачных вычислений.

В главе 14, «Оптимизация на уровне приложения», мы объясним то, что называется оптимизацией во всем стеке, — оптимизацию различных элементов, от фронтенда до бэкенда, от пользовательского интерфейса до базы данных.

Хорошо спроектированная масштабируемая база данных должна быть защищена от сбоев электроснабжения, атак злоумышленников, ошибок в приложениях и прочих напастей. В главе 15, «Резервное копирование и восстановление», мы обсудим различные стратегии резервного копирования и восстановления баз данных MySQL. Эти стратегии помогут минимизировать время простоя в случае выхода из строя оборудования и гарантировать, что данные переживут такую катастрофу.

Различные полезные темы

В последней главе и приложениях мы углубимся в вопросы, которые либо не вписываются ни в одну из предыдущих глав, либо так часто упоминаются в них, что заслуживают отдельного рассмотрения.

В главе 16, «Инструменты для пользователей MySQL», описаны коммерческие инструменты, а также инструменты с открытым исходным кодом, которые можно использовать для более эффективного управления серверами MySQL и их мониторинга.

В приложении А рассмотрены три основные неофициальные версии MySQL, появившиеся за последние несколько лет, в том числе та, которую поддерживает наша компания. Стоит знать, что еще можно использовать при необходимости: многие задачи, плохо поддающиеся решению или вообще неразрешимые с помощью MySQL, легко решаются с помощью одной из этих версий. Две из трех, Percona Server и MariaDB, незначительно отличаются от MySQL, поэтому их внедрение потребует небольших усилий. Тем не менее стоит добавить, что официального дистрибутива MySQL от Oracle вполне достаточно для большинства пользователей.

В приложении Б показано, как разобраться в текущем режиме работы сервера MySQL. Очень важно знать, как получить информацию о состоянии сервера. Но еще важнее понимать, что эта информация означает. Мы подробно рассмотрим команду `SHOW INNODB STATUS`, поскольку она позволяет детально разобраться в операциях, осуществляемых транзакционной подсистемой хранения InnoDB. Внутреннее устройство InnoDB также обсуждается в этом приложении.

В приложении В говорится о том, как эффективно копировать очень большие файлы, что критически важно при работе со значительными объемами данных. В приложении Г показано, как на практике использовать очень полезную команду `EXPLAIN`. Приложение Д поможет вам выяснить, что происходит, когда запросы приводят к конфликтующим друг с другом блокировкам. И наконец, приложение Е представляет собой введение в высокопроизводительную систему полнотекстового поиска Sphinx, которая дополняет возможности MySQL.

Версии программного обеспечения и их доступность

MySQL постоянно меняется. С тех пор как Джереми набросал план первого издания этой книги, появилось множество версий MySQL. Когда первое издание готовилось к печати, MySQL 4.1 и 5.0 существовали только в виде альфа-версий, а сейчас MySQL 5.1 и 5.5 уже стали основой крупных веб-приложений. На момент окончания подготовки третьего издания MySQL 5.6 еще не была выпущена, но ожидалось, что она станет ультрасовременной.

В этой книге мы не ограничиваемся какой-то конкретной версией, а опираемся на свой обширный опыт работы с MySQL в реальных приложениях. В основном речь

идет о версиях MySQL 5.1 и 5.5, поскольку именно их мы считаем текущими. В большинстве примеров предполагается, что вы используете какую-то относительно зрелую версию MySQL 5.1, например MySQL 5.1.50 или более новую. Мы старались отмечать возможности, которые отсутствуют в более старых версиях или появятся только в следующем семействе 5.6. Однако авторитетным источником информации о возможностях каждой конкретной версии является сама документация по MySQL. Мы надеемся, что в процессе чтения этой книги вы будете время от времени посещать сайт разработчиков СУБД, содержащий всю необходимую информацию (http://dev.mysql.com/doc/).

Другим замечательным свойством MySQL является то, что она работает практически на всех современных платформах: Mac OS X, Windows, GNU/Linux, Solaris, FreeBSD и других! Однако мы предпочитаем GNU/Linux[1] и иные UNIX-подобные операционные системы. Пользователи Windows, вероятно, обнаружат здесь некоторые различия. Например, пути к файлам записываются совершенно иначе. Мы также ссылаемся на стандартные утилиты командной строки UNIX и предполагаем, что вы знаете соответствующие команды в Windows[2].

Еще одна трудность при работе с MySQL на платформе Windows — отсутствие языка Perl в стандартной поставке операционной системы. В состав дистрибутива MySQL входят несколько полезных утилит, написанных на Perl. В отдельных главах этой книги представлены примеры Perl-скриптов: они служат основой для более сложных инструментов, которые создадите уже вы. Комплект Percona Toolkit также написан на Perl. Чтобы использовать эти скрипты, вам потребуется загрузить версию Perl для Windows с сайта компании ActiveState и установить дополнительные модули (DBI и DBD::mysql) для доступа к MySQL.

Условные обозначения

В книге применяются следующие условные обозначения.

Курсив

Курсивом выделяются новые термины, слова, на которых сделан акцент.

Шрифт для названий

Применяется для отображения URL, адресов электронной почты.

`Моноширинный шрифт`

Применяется для обозначения конфигурационных параметров, имен таблиц, имен и значений переменных, имен функций, модулей, имен файлов и путей к ним, названий

[1] Во избежание путаницы мы ссылаемся на Linux, когда пишем о ядре, и на GNU/Linux, когда пишем обо всей инфраструктуре операционной системы, поддерживающей приложения.

[2] Вы можете найти версии UNIX-утилит для Windows на сайтах http:// unxutils.sourceforge.net или http://gnuwin32.sourceforge.net.

каталогов, команд и утилит UNIX, содержимого файлов и результатов работы команд, имен пользователей и хостов. Кроме того, он применяется для фрагментов кода.

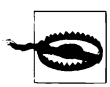
Моноширинный полужирный шрифт

Применяется для команд или другого текста, который пользователь должен ввести дословно. Им также выделены результаты работы команды.

Моноширинный курсив

Им выделен текст, вместо которого пользователь должен подставить значения по контексту.

В такой врезке представлены советы, предложения и примечания общего характера.



Таким способом выделяются предупреждения и предостережения.

О примерах кода

Задача этой книги — помочь вам делать вашу работу. Вы можете использовать любой пример кода из книги в ваших программах и документации. Обращаться к нам за разрешением нет необходимости, если только вы не копируете значительную часть кода. Например, написание программы с использованием нескольких фрагментов кода из этой книги не требует отдельного разрешения. Однако для продажи или распространения компакт-диска с примерами из книг O'Reilly разрешение необходимо. Ответ на вопрос путем цитирования этой книги и цитирования примеров кода не требует разрешения. Но для включения значительного количества примеров кода из книги в документацию к вашему продукту нужно разрешение.

Примеры можно найти на сайте http://www.highperfmysql.com, где они периодически обновляются. Однако мы не в состоянии обновлять и тестировать код для каждой версии MySQL.

Мы ценим, хотя и не требуем ссылки на первоисточник. Такая ссылка включает название, автора, издательство и ISBN. Например: High Performance MySQL, Third Edition, by Baron Schwartz et al. (O'Reilly). Copyright 2012 Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko, 978-1-449-31428-6.

Если вам кажется, что вы выходите за рамки правомерного использования примеров кода, связывайтесь с нами по адресу permissions@oreilly.com.

Благодарности к третьему изданию

Выражаем благодарность всем тем, кто нам помогал: Брайану Акеру (Brian Aker), Йохану Андерссону (Johan Andersson), Эспену Браккену (Espen Braekken), Марку Каллагану (Mark Callaghan), Джеймсу Дэю (James Day), Мацею Добржанскому (Maciej Dobrzanski), Эвену Форчуну (Ewen Fortune), Дэйву Хильдебрандту (Dave Hildebrandt), Фернандо Ипару (Fernando Ipar), Хайдуну Джи (Haidong Ji), Джузеппе Максии (Giuseppe Maxia), Ауримасу Микалаускасу (Aurimas Mikalauskas), Иштвану Подору (Istvan Podor), Иву Трюдо (Yves Trudeau), Мэтту Йонковиту (Matt Yonkovit) и Алексу Юрченко (Alex Yurchenko). Спасибо всем в Percona за то, что на протяжении многих лет они помогали нам, каждый по-своему. Хотим поблагодарить множество замечательных блогеров[1] и докладчиков, которые постоянно давали нам пищу для размышлений, в особенности Йошинори Мацунобу (Yoshinori Matsunobu). Спасибо также авторам предыдущих изданий: Джереми Д. Заводны (Jeremy D. Zawodny), Дереку Дж. Баллингу (Derek J. Balling) и Арьену Ленцу (Arjen Lentz). Также благодарим Энди Орама (Andy Oram), Рейчел Хед (Rachel Head) и всех сотрудников издательства O'Reilly, которые так классно издают книги и проводят конференции. Еще хотим сказать большое спасибо блестящей и преданной делу команде MySQL, работающей в Oracle, а также всем тем, кто ранее трудился над MySQL, где бы вы ни были, и особенно тем, кто был связан со SkySQL и Monty Program.

Бэрон хочет поблагодарить жену Линн, свою мать Конни, а также тещу и тестя, Джейн и Роджера, за то, что они всеми силами поддерживали его при работе над этим проектом, ободряли, помогали в делах и заботились о семье. Большое спасибо также Петру и Вадиму — вы замечательные учителя и коллеги. Бэрон посвящает это издание памяти Алана Римм-Кауфмана, чьи великую любовь и поддержку он всегда будет помнить.

Благодарности ко второму изданию

Разработчик системы Sphinx Андрей Аксенов (Andrew Aksyonov) написал приложение E «Использование Sphinx совместно с MySQL». Мы ставим Андрея на первое место в перечне благодарностей за активное участие в обсуждениях.

В процессе написания этой книги мы получили бесценную помощь многих людей. Невозможно перечислить всех, кто нам помогал, — мы должны поблагодарить все сообщество MySQL и каждого сотрудника компании MySQL AB. Однако вот список тех, кто внес непосредственный вклад в создание второго издания (просим извинить, если мы кого-то пропустили): Тобиас Асплунд (Tobias Asplund), Игорь Бабаев (Igor Babaev), Паскаль Боргино (Pascal Borghino), Роланд Боуман (Roland Bouman), Рональд Брэдфорд (Ronald Bradford), Марк Каллаган (Mark Callaghan), Джереми Коул (Jeremy Cole), Бритт Кроуфорд (Britt Crawford) и проект HiveDB, Васил Димов (Vasil Dimov), Харрисон Фиск (Harrison Fisk), Флориан Хаас (Florian

[1] На сайте http://planet.mysql.com можно найти множество замечательных технических блогов.

Haas), Дмитрий Жуковский (Dmitri Joukovski) и Zmanda (благодарим за схему, поясняющую мгновенные снимки LVM), Алан Казиндорф (Alan Kasindorf), Шеери Критцер Кабрал (Sheeri Kritzer Cabral), Марко Макела (Marko Makela), Джузеппе Максиа (Giuseppe Maxia), Пол Маккалаг (Paul McCullagh), Б. Кит Мэрфи (B. Keith Murphy), Дирен Пэйтел (Dhiren Patel), Сергей Петруня (Sergey Petrunia), Александр Рубин (Alexander Rubin), Пол Такфилд (Paul Tuckfield), Хейкки Туури (Heikki Tuuri) и Майкл «Монти» Видениус (Michael "Monty" Widenius).

Особая благодарность Энди Ораму (Andy Oram) и Изабель Канкл (Isabel Kunkle), редактору и помощнику редактора нашей книги из издательства O'Reilly, а также Рейчел Вилер (Rachel Wheeler), литературному редактору. Благодарим и остальных сотрудников издательства O'Reilly.

От Бэрона

Я хочу поблагодарить свою жену Линн Рейнвилл и нашего пса по кличке Карбон. Если вам доводилось писать книгу, то, уверен, вы можете представить, как бесконечно я им благодарен. Я также благодарю Алана Римм-Кауфмана (Alan Rimm-Kaufman) и своих коллег из компании Rimm-Kaufman Group за то, что они поддерживали и ободряли меня в ходе работы над этим проектом. Спасибо Петру, Вадиму и Арьену за то, что они дали мне возможность реализовать свою мечту. И спасибо Джереми и Дереку, которые проложили нам путь.

От Петра

Я годами занимался созданием презентаций, обучением и консультированием по вопросам производительности и масштабирования MySQL и всегда хотел иметь более широкую аудиторию, поэтому был очень взволнован, когда Энди Орам предложил мне поработать над этой книгой. Прежде мне не приходилось писать книг, поэтому я не ожидал, что это займет столько времени и потребует таких усилий. Сначала мы собирались лишь обновить первое издание с учетом последних версий MySQL, но оказалось, что надо добавить так много материала, что было решено переписать почти всю книгу.

Это издание является результатом усилий целой команды. Поскольку я был очень загружен раскруткой нашей с Вадимом консультационной компании Percona, а английский для меня не родной язык, то у всех нас были разные роли. Я предложил план и подобрал технический материал, а затем по мере написания книги занимался пересмотром и расширением этого материала. Когда к проекту присоединился Арьен (бывший руководитель группы подготовки документации MySQL), мы приступили к реализации плана. Дело действительно закрутилось, когда к нам присоединился Бэрон, который способен писать высококачественный текст с бешеной скоростью. Вадим оказал огромную помощь в тщательной проверке исходного кода MySQL, а также подготовке эталонных тестов и проведении других исследований.

По мере работы над книгой мы обнаруживали, что хотим более детально рассмотреть все больше и больше областей. Многие темы, например репликация, оптимизация

запросов, архитектура InnoDB и проектирование, вполне заслуживают отдельных книг, поэтому нам приходилось в какой-то момент останавливаться, оставляя часть материала для возможного следующего издания или наших блогов, презентаций и статей.

Мы получили неоценимую помощь от рецензентов — ведущих экспертов по MySQL, работающих как в MySQL AB, так и в других местах. В их числе создатель MySQL Майкл Видениус, автор InnoDB Хейкки Туури, руководитель группы разработчиков оптимизатора MySQL Игорь Бабаев и многие другие.

Я хочу поблагодарить свою жену Катю Зайцеву и детей Ваню и Надю за то, что они с пониманием отнеслись к необходимости заниматься книгой, из-за чего порой я не мог уделять им достаточно времени. Я также благодарен сотрудникам компании Percona, которые подменяли меня, когда я работал над рукописью. Отдельное спасибо хочется сказать Энди Ораму и издательству O'Reilly за то, что книга наконец увидела свет.

От Вадима

Я хочу поблагодарить Петра, вдохновившего меня потрудиться над этой книгой, Бэрона, сыгравшего важную роль в том, что она была написана, и Арьена, работать с которым было очень весело. Спасибо также нашему редактору Энди Ораму, проявившему изрядное терпение по отношению к нам, команде MySQL, создавшей прекрасный продукт, и нашим клиентам, благодаря которым у меня появилась возможность хорошо разобраться в этой СУРБД. И наконец, особая благодарность моей жене Валерии и нашим сыновьям Мирославу и Тимуру, которые всегда поддерживали меня и помогали двигаться вперед.

От Арьена

Я хочу поблагодарить Энди за его мудрость, терпение и за то, что направлял нас. Спасибо Бэрону за то, что он запрыгнул в поезд второго издания, когда тот уже набирал ход, Петру и Вадиму — за базовую информацию и тестирование. Спасибо также Джереми и Дереку за первое издание. Как ты, Дерек, написал мне: «Все, что я прошу: не давай им расслабиться».

Хочу сказать спасибо всем моим бывшим коллегам (и нынешним друзьям) из MySQL AB, где я получил большую часть знаний на эту тему. В данном контексте особо хочу упомянуть Монти, которого продолжаю считать отцом MySQL даже несмотря на то, что его компания теперь является частью Sun Microsystems. Также я хочу выразить благодарность всем остальным участникам глобального сообщества MySQL.

Напоследок благодарю свою дочь Фебу, которая в своем столь юном возрасте еще не интересуется вещью под названием MySQL. Для некоторых неведение является настоящим благом, и они показывают нам, что действительно важно в нашем мире. Для остальных же эта книга может стать полезным пополнением. И не забывайте о своей жизни.

Благодарности к первому изданию

Появление книг, подобных этой, невозможно без участия десятков людей. Без их помощи издание, которое вы держите в руках, скорее всего, осталось бы кучей записок, прилепленных к нашим мониторам. Здесь мы хотим сказать теплые слова о тех, кто нам помогал (хорошо, что нам не приходится беспокоиться о фоновой музыке, призывающей нас замолчать и уйти, как это иногда бывает во время вручения разных премий).

Мы не могли бы закончить этот проект без постоянных просьб, подталкивания и поддержки со стороны нашего редактора Энди Орама. Если нужно назвать основного человека, чьей заслугой является выход этой книги, то это Энди. Мы действительно ценим еженедельные совещания с ним, которые не давали нам расслабиться.

Однако Энди не единственный. В издательстве O'Reilly много людей, принимавших участие в превращении наших заметок в книгу, которую вам так хочется прочитать. Мы также хотим поблагодарить людей, занимавшихся ее иллюстрированием, производством и продвижением. И конечно, спасибо Тиму О'Рейли за постоянную заботу о создании прекрасной документации для популярных программных продуктов с открытым кодом.

Наконец, мы оба хотим сказать большое спасибо рецензентам, которые согласились просматривать черновики этой книги и говорили нам о том, что мы делали неправильно. Они потратили часть своего отпуска в 2003 году на просмотр первых версий этого текста, полного опечаток, вводящих в заблуждение выражений и откровенных математических ошибок. Не выделяя никого в отдельности, мы благодарим Брайана «Krow» Алкера (Brian «Krow» Aker), Марка «JDBC» Мэтьюса (Mark «JDBC» Matthews), Джереми «the other Jeremy» Коула (Jeremy «the other Jeremy» Cole), Майка «VBMySQL.com» Хилльера (Mike «VBMySQL.com» Hillyer), Реймонда «Rainman» Де Po (Raymond «Rainman» De Roo), Джеффри «Regex Master» Фридла (Jeffrey «Regex Master» Friedl), Джейсона Дехаана (Jason DeHaan), Дена Нелсона (Dan Nelson), Стива «UNIX Wiz» Фридла (Steve «UNIX Wiz» Friedl) и Касию «UNIX Girl» Трапзо (Kasia «UNIX Girl» Trapszo).

От Джереми

Я снова хочу поблагодарить Энди за его согласие взяться за этот проект и неизменную поддержку. Помощь Дерека была существенной в процессе подготовки последних 20–30 % книги: не будь ее, мы бы не уложились в отведенные сроки. Спасибо ему за согласие принять участие в проекте на поздней стадии, работу над разделом, посвященным поддержке XML, главой 10, приложением E и другими частями книги.

Я также хочу поблагодарить своих родителей за то, что они много лет назад подарили мне мой первый компьютер Commodore 64. Они не только терпели первые десять лет компьютерного фанатизма, но и быстро стали помощниками в моих непрекращающихся поисках новых знаний.

Кроме того, я хочу поблагодарить группу людей, от работы с которыми по вербовке приверженцев MySQL на Yahoo! я получал удовольствие в последние несколько лет. Джеффри Фридл и Рей Голдбергер вдохновляли меня и помогали мне на первых этапах этого предприятия. Стив Моррис, Джеймс Харви и Сергей Колычев участвовали в моих экспериментах с серверами MySQL на Yahoo! Finance, даже когда это отвлекало их от важных дел. Также благодарю бесчисленное множество других пользователей Yahoo!, помогавших мне находить интересные задачи и решения, относящиеся к MySQL. И, что важнее всего, спасибо им за веру в меня, благодаря которой MySQL стала одной из самых важных и заметных частей бизнеса Yahoo!.

Адам Гудман, издатель и владелец Linux Magazine, помог мне начать писать для тех, кто нуждается в технической литературе, опубликовав мои статьи по MySQL в 2001 году. Он даже сам не понимает, сколь многому научил меня в плане редактирования и издательского дела. Именно он вдохновил меня не сворачивать с этого пути и продолжать вести ежемесячную колонку в журнале. Спасибо, Адам.

Спасибо Монти и Дэвиду за распространение MySQL по всему миру. И раз уж я заговорил о компании MySQL AB, спасибо замечательным людям, вдохновлявшим меня на написание этой книги: Керри, Лари, Джо, Мартену, Брайану, Полу, Джереми, Марку, Харрисону, Мэтту и всей остальной команде.

Наконец, спасибо всем читателям моего блога за ежедневное неформальное общение на тему MySQL и обсуждение других технических вопросов. И спасибо Гун Сквад.

От Дерека

Как и Джереми, я должен поблагодарить свою семью, в основном по тем же самым причинам. Я хочу выразить благодарность своим родителям за то, что они постоянно подталкивали меня к написанию книги. Бабушка и дедушка одолжили мне денег на покупку первого компьютера Commodore VIC-20, и в результате я понял важность долларов и влюбился в компьютеры.

Я не могу выразить всю свою благодарность Джереми за то, что он пригласил меня для совместной работы над этой книгой. Это прекрасный опыт, и я был бы рад снова поработать с ним.

Особое спасибо Реймонду Де Ро (Raymond De Roo), Брайану Вольгемуту (Brian Wohlgemuth), Дэвиду Калафранческо (David Calafrancesco), Тере Доти (Tera Doty), Джею Рубину (Jay Rubin), Биллу Катлану (Bill Catlan), Энтони Хоуву (Anthony Howe), Марку О'Нилу (Mark O'Neal), Джорджу Монтгомери (George Montgomery), Джорджу Барберу (George Barber) и множеству других, которые терпеливо выслушивали меня, старались понять, что я хочу сказать, или просто заставляли улыбнуться, когда я больше всего нуждался в этом. Без вас я до сих пор писал бы эту книгу и почти наверняка спятил бы в процессе.

1 История и архитектура MySQL

Архитектура MySQL очень отличается от архитектур иных серверов баз данных, что делает эту СУБД полезной для одних целей, но одновременно неудачным выбором для других. MySQL неидеальна, но достаточно гибка для того, чтобы хорошо работать в очень требовательных средах, например в веб-приложениях. В то же время MySQL позволяет применять встраиваемые приложения, хранилища данных, индексирование содержимого, программное обеспечение для доставки, высоконадежные системы с резервированием, обработку транзакций в реальном времени (OLTP) и многое другое.

Для того чтобы максимально эффективно использовать MySQL, нужно разобраться в ее устройстве. Гибкость системы проявляется во многом. Например, вы можете настроить ее для работы на различном оборудовании и поддержки разных типов данных. Однако самой необычной и важной особенностью MySQL является такая архитектура подсистемы хранения, в которой обработка запросов и другие серверные задачи отделены от хранения и извлечения данных. Подобное разделение задач позволяет выбирать способ хранения данных, а также настраивать производительность, ключевые характеристики и др.

В текущей главе сделан краткий обзор архитектуры сервера MySQL, рассматриваются основные различия между подсистемами хранения и говорится о том, почему эти различия важны. В конце главы мы рассмотрим исторический контекст и перечислим эталонные тесты производительности (бенчмарки). Попытаемся объяснить MySQL на примерах и максимально упрощая детали. Этот обзор будет полезен как для новичков в работе с сервером баз данных, так и для читателей, которые являются экспертами в работе с подобными системами.

Логическая архитектура MySQL

Чтобы хорошо понимать работу сервера, нужно иметь представление о взаимодействии его компонентов. На рис. 1.1 представлен логический вид архитектуры MySQL.

Рис. 1.1. Логический вид архитектуры сервера MySQL

На верхнем уровне располагаются службы, не являющиеся уникальными компонентами MySQL. Они необходимы большинству сетевых клиент-серверных инструментов или серверов: для обслуживания соединений, аутентификации, обеспечения безопасности и т. п.

Второй уровень намного интереснее. Здесь находится бóльшая часть «мозгов» MySQL: код для обработки, анализа, оптимизации и кэширования запросов, а также все встроенные функции (например, функции даты/времени, математические и функции шифрования). Здесь также расположены все инструменты, используемые в подсистемах хранения, например хранимые процедуры, триггеры и представления.

Третий уровень содержит подсистемы хранения данных. Они отвечают за хранение всех данных в MySQL и их извлечение. Подобно различным файловым системам, доступным для GNU/Linux, каждая подсистема хранения данных имеет как сильные, так и слабые стороны. Сервер взаимодействует с ними через API подсистемы хранения данных. Этот интерфейс скрывает различия между такими подсистемами и делает их практически прозрачными на уровне запросов. API содержит пару десятков низкоуровневых функций, выполняющих операции типа «начать транзакцию» или «извлечь строку с таким первичным ключом». Подсистемы хранения не анализируют запросы SQL[1] и не взаимодействуют друг с другом, они просто отвечают на исходящие от сервера запросы.

[1] Единственным исключением является InnoDB, где анализируются определения внешнего ключа, поскольку сам сервер MySQL не делает этого.

Управление соединениями и их безопасность

Для каждого клиентского соединения внутри серверного процесса выделяется отдельный поток. Запросы соединения выполняются только внутри этого потока, который, в свою очередь, выполняется одним ядром или процессором. Сервер кэширует потоки, поэтому их не нужно создавать или уничтожать для каждого нового соединения[1].

Когда клиенты (приложения) подключаются к серверу MySQL, он должен их аутентифицировать. Аутентификация выполняется на основе имени пользователя, адреса хоста, с которого происходит соединение, и пароля. При соединении по протоколу Secure Sockets Layer (SSL) можно использовать сертификаты X.509. После того как клиент подключился, сервер проверяет наличие необходимых привилегий для каждого запроса (например, может ли клиент использовать команду `SELECT` применительно к таблице `Country` базы данных `world`).

Оптимизация и выполнение

MySQL анализирует запросы для создания внутренней структуры (дерева разбора), а затем выполняет оптимизации. Это могут быть переписывание запроса, определение порядка чтения таблиц, выбор используемых индексов и т. п. Через специальные ключевые слова в запросе вы можете передать оптимизатору подсказки и тем самым повлиять на процесс принятия решения. Или обратиться к серверу за объяснением различных аспектов оптимизации. Это позволит вам понять, какие решения принимает сервер, и даст ориентир для изменения запросов, схем и настроек, чтобы добиться максимальной эффективности работы. Оптимизатор мы детально обсудим в главе 6.

Оптимизатору неважно, в какой подсистеме хранения данных находится конкретная таблица, но подсистема хранения данных влияет на то, как сервер оптимизирует запрос. Оптимизатор опрашивает подсистему хранения данных о некоторых ее возможностях, затратах на выполнение определенных операций и статистике по содержащимся в таблицах данным. Например, отдельные подсистемы хранения поддерживают типы индексов, которые могут быть полезны для выполнения определенных запросов. Больше информации об индексировании и схемах оптимизации вы сможете найти в главах 4 и 5.

Прежде чем анализировать запрос, сервер обращается к кэшу запросов, в котором могут храниться только команды `SELECT` вместе с наборами результатов. Если поступает запрос, идентичный уже имеющемуся в кэше, серверу не нужно выполнять анализ, оптимизацию или сам запрос — он может просто отправить в ответ сохраненный набор результатов. Больше об этом можно узнать, прочитав главу 7.

---

[1] MySQL 5.5 и более новые версии поддерживают API, который может принимать плагины для организации пула потоков, поэтому небольшой пул потоков может обслуживать множество соединений.

Управление конкурентным доступом

Задача управления конкурентным доступом возникает в тот момент, когда нескольким запросам необходимо одновременно изменить данные. В рамках текущей главы оговорим, что MySQL должна решать эту задачу на двух уровнях: сервера и подсистемы хранения данных. Управление конкурентным доступом — это обширная тема, которой посвящено множество теоретических исследований. Мы же просто представим обзор того, что MySQL делает с конкурентными запросами на чтение и запись, чтобы вы смогли получить общее представление об этой теме, а это, в свою очередь, позволит разобраться в материале главы.

В качестве примера будем использовать ящик электронной почты в системе UNIX. Классический файл формата mbox очень прост. Все сообщения в почтовом ящике mbox расположены одно за другим, так что читать и анализировать почтовые сообщения очень просто. Это также существенно упрощает доставку почты: достаточно добавить новое сообщение в конец файла.

Но что происходит, когда два процесса пытаются одновременно поместить сообщения в почтовый ящик? Очевидно, что чередование строк этих сообщений приведет к повреждению файла. Чтобы предотвратить это, правильно работающие почтовые системы используют блокировку. Если клиент пытается отправить новое сообщение в тот момент, когда почтовый ящик заблокирован, ему придется подождать, пока он не сможет сам использовать блокировку, чтобы отправить сообщение.

Эта схема довольно хорошо работает, но не поддерживает конкурентный доступ. Поскольку в любой момент времени только один процесс может изменять содержимое почтового ящика, такой подход создает проблемы при работе с большими почтовыми ящиками.

Блокировки чтения/записи

Чтение из почтового ящика не вызывает таких проблем. Ничего страшного, если несколько клиентов одновременно считывают информацию из одного и того же почтового ящика. Раз они не вносят изменений, ничего плохого случиться не должно. Но что произойдет, если кто-нибудь попытается удалить сообщение № 25 в тот момент, когда программы читают письма из почтового ящика? Возможны различные варианты развития ситуации, но программа чтения может получить почтовый ящик в поврежденном или неструктурированном виде. Поэтому для обеспечения безопасности даже чтение информации из почтового ящика требует определенных предосторожностей.

Если представить, что почтовый ящик — это таблица базы данных, а каждое почтовое сообщение — строка, легко увидеть, что и в этом контексте актуальна та же проблема. Во многих смыслах почтовый ящик является простой таблицей базы данных. Модификация строк в такой базе очень похожа на удаление или изменение содержимого сообщений в файле почтового ящика.

У классической задачи управления конкурентным доступом довольно простое решение. В системах, которые имеют дело с конкурентным доступом для чтения/записи, чаще всего реализуется система блокирования, содержащая два типа блокировок. Обычно их называют *разделяемыми блокировками* и *монопольными блокировками*, или *блокировками чтения* и *блокировками записи*.

Не вдаваясь в подробности технологии блокирования, данную концепцию можно описать следующим образом. Блокировки чтения ресурса являются разделяемыми или взаимно неблокирующими: множество клиентов могут производить считывание из ресурса в одно и то же время, не мешая друг другу. Блокировки записи, напротив, являются эксклюзивными. Другими словами, они исключают возможность установки блокировки чтения и других блокировок записи, поскольку единственной безопасной политикой является наличие только одного клиента, выполняющего запись в данный момент времени, и предотвращение во время этого всех операций чтения.

В базах данных блокировки происходят постоянно: MySQL запрещает одному клиенту считывать данные, когда другой клиент их изменяет. Управление блокировками осуществляется внутри СУБД в соответствии с принципами, которые достаточно прозрачны для клиентов.

Детальность блокировок

Одним из способов улучшения конкурентного доступа к разделяемому ресурсу является увеличение избирательности блокировок. Вместо того чтобы блокировать весь ресурс, можно заблокировать только ту его часть, которая содержит изменяемые данные. Еще лучше заблокировать лишь тот фрагмент данных, который будет изменен. Минимизация объема данных, которые вы блокируете в каждый момент времени, позволяет одновременно выполнять несколько изменений одного и того же ресурса, если эти операции не конфликтуют друг с другом.

Определенная проблема возникает из-за того, что блокировки потребляют ресурсы. Каждая операция блокирования — получение возможности блокировки, проверка того, можно ли применить блокировку, снятие блокировки и т. п. — влечет за собой издержки. Если система тратит слишком много времени на управление блокировками вместо того, чтобы расходовать его на сохранение и извлечение данных, то это может повлиять на производительность.

Стратегия блокирования является компромиссом между неизбежностью издержек на реализацию блокировок и безопасностью данных, причем подобный компромисс влияет на производительность. Большинство коммерческих серверов баз данных не предоставляют особого выбора: вы получаете возможность блокировки таблиц на уровне строки, при этом нередко в сочетании с множеством сложных способов обеспечить хорошую производительность при большом количестве блокировок.

MySQL также предоставляет выбор. Подсистемы хранения данных MySQL могут реализовывать собственные стратегии блокировки и уровни детализации блокировок. Управление блокировками является очень важным решением при проектировании подсистем хранения данных. Установка детализации на определенном уровне в ряде случаев может улучшить производительность, но сделать эту подсистему менее подходящей для других целей. Поскольку MySQL предлагает несколько подсистем хранения данных, нет необходимости принимать единственное решение на все случаи жизни. Рассмотрим две наиболее важные стратегии блокировок.

Табличные блокировки

Табличная блокировка является базовой стратегией блокировки в MySQL. Кроме того, у нее самые низкие издержки. Табличная блокировка аналогична рассмотренным ранее блокировкам почтового ящика — блокируется вся таблица. Когда клиент хочет сделать запись в таблицу (вставку, удаление, обновление и т. п.), он получает блокировку на запись для всей таблицы. Это предотвращает все остальные операции чтения и записи. Когда никто не выполняет запись, любой клиент может получить блокировку на чтение, которая не конфликтует с другими подобными блокировками.

У табличных блокировок есть вариации для обеспечения высокой производительности в различных ситуациях. Например, табличные блокировки `READ LOCAL` разрешают выполнять некоторые типы параллельных операций записи. Кроме того, блокировки записи имеют более высокий приоритет, чем блокировки чтения. Поэтому запрос блокировки записи будет помещен в очередь перед уже находящимися там запросами блокировки чтения (блокировки записи могут отодвигать в очереди блокировки чтения, а блокировки чтения не могут отодвигать блокировки записи).

Подсистемы хранения также могут управлять собственными блокировками.

MySQL использует множество блокировок, эффективных на уровне таблиц, для различных целей. Например, для таких операторов, как `ALTER TABLE`, сервер применяет табличную блокировку вне зависимости от подсистемы хранения данных.

Построчные блокировки

Построчные блокировки обеспечивают лучшее управление конкурентным доступом (и влекут максимальные издержки). Блокировка на уровне строк доступна, в частности, в подсистемах хранения InnoDB и XtraDB. Построчные блокировки реализуются подсистемами хранения данных, а не сервером (если нужно, еще раз взгляните на рис. 1.1). Сервер ничего не знает о блокировках, реализованных подсистемой хранения данных, и, как вы увидите далее, все подсистемы хранения данных реализуют блокировки по-своему.

Транзакции

До тех пор пока не познакомитесь с *транзакциями*, вы не сможете изучать более сложные функции СУБД.

Транзакция представляет собой группу запросов SQL, обрабатываемых *атомарно*, то есть как единое целое. Если подсистема базы данных может выполнить всю группу запросов, она делает это, но если какой-либо запрос не может быть выполнен в результате сбоя или по иной причине, ни один запрос группы не будет выполнен. Все или ничего.

Немногое в этом разделе характерно именно для MySQL. Если вы уже знакомы с транзакциями ACID, можете спокойно перейти к подразделу «Транзакции в MySQL».

Банковское приложение является классическим примером, демонстрирующим необходимость транзакций. Представьте банковскую базу данных с двумя таблицами: `checking` (текущие счета) и `savings` (сберегательные счета). Чтобы перевести 200 долларов с текущего счета Джейн на ее сберегательный счет, вам нужно сделать по меньшей мере три шага.

1. Убедиться, что остаток на ее текущем счете больше 200 долларов.
2. Вычесть 200 долларов из остатка текущего счета.
3. Добавить 200 долларов к остатку сберегательного счета.

Вся операция должна быть организована как транзакция, чтобы в случае неудачи на любом из трех этапов все выполненные ранее шаги были отменены.

Вы начинаете транзакцию командой `START TRANSACTION`, а затем либо сохраняете изменения командой `COMMIT`, либо отменяете их командой `ROLLBACK`. Код SQL для транзакции может выглядеть следующим образом:

```
1 START TRANSACTION;
2 SELECT balance FROM checking WHERE customer_id = 10233276;
3 UPDATE checking SET balance = balance - 200.00 WHERE customer_id = 10233276;
4 UPDATE savings  SET balance = balance + 200.00 WHERE customer_id = 10233276;
5 COMMIT;
```

Но сами по себе транзакции — это еще не все. Что произойдет в случае сбоя сервера базы данных во время выполнения четвертой строки? Кто знает... Клиент, вероятно, потеряет 200 долларов. А если другой процесс вклинится между выполнением строк 3 и 4 и снимет весь остаток с текущего счета? Банк предоставит клиенту кредит 200 долларов, даже не зная об этом.

Транзакций недостаточно, пока система не прошла *тест ACID*. Аббревиатура ACID расшифровывается как atomicity, consistency, isolation и durability (атомарность, согласованность, изолированность и долговечность). Это тесно связанные критерии,

которым должна соответствовать правильно функционирующая система обработки транзакций.

- **Атомарность.** Транзакция должна функционировать как единая неделимая рабочая единица таким образом, чтобы вся она была либо выполнена, либо отменена. Для атомарных транзакций не существует такого понятия, как частичное выполнение: все или ничего.
- **Согласованность.** База данных всегда должна переходить из одного согласованного состояния в другое. В нашем примере согласованность гарантирует, что сбой между строками 3 и 4 не приведет к исчезновению с текущего счета 200 долларов. Поскольку транзакция не будет подтверждена, ни одно из изменений не отразится в базе данных.
- **Изолированность.** Результаты транзакции обычно невидимы другим транзакциям, пока она не подтверждена. В нашем примере это гарантирует, что, если программа суммирования остатков на банковских счетах будет запущена после третьей строки перед четвертой, она по-прежнему увидит 200 долларов на текущем счете. Когда будем рассматривать уровни изолированности, вы поймете, почему здесь сказано «обычно невидимы».
- **Долговечность.** После подтверждения внесенные в ходе транзакции изменения становятся постоянными. Это значит, что они должны быть записаны так, чтобы данные не потерялись при сбое системы. Долговечность, однако, является несколько расплывчатой концепцией, поскольку у нее довольно много уровней. Некоторые стратегии обеспечения долговечности дают более высокие гарантии безопасности, чем другие, и ни одна из них не является надежной на 100 % (если база данных долговечна сама по себе, то каким образом резервное копирование повышает долговечность?). В последующих главах мы еще обсудим, что же на самом деле в MySQL означает долговечность.

Транзакции ACID гарантируют, что банк не потеряет ваши деньги. Вообще очень сложно, а то и невозможно сделать это с помощью логики приложения. Сервер базы данных, поддерживающий ACID, должен выполнить множество сложных операций, о которых вы, возможно, даже не подозреваете, чтобы обеспечить гарантии ACID.

Как и в случае увеличения детализации блокировок, оборотной стороной усиленной безопасности является увеличение объема работы сервера базы. Сервер базы данных с транзакциями ACID также требует бóльших мощности процессора, объема памяти и дискового пространства, чем сервер без них. Как мы уже отмечали, это тот самый случай, когда архитектура подсистем хранения данных MySQL является вашим союзником. Вы сами можете решить, требует ли приложение использования транзакций. Если они не нужны, вы можете добиться большей производительности, выбрав для некоторых типов запросов нетранзакционную подсистему хранения данных. С помощью команды `LOCK TABLES` можно установить нужный уровень защиты без использования транзакций. Все в ваших руках.

Уровни изолированности

Изолированность — более сложное понятие, чем кажется на первый взгляд. Стандарт SQL определяет четыре уровня изолированности с конкретными правилами, устанавливающими, какие изменения видны внутри и за пределами транзакции, а какие — нет. Более низкие уровни изолированности обычно допускают бóльшую степень конкурентного доступа и влекут за собой меньшие издержки.

Все подсистемы хранения данных реализуют уровни изолированности немного по-разному, и они не всегда будут соответствовать вашим ожиданиям, если вы привыкли к другой СУБД (здесь не будем вдаваться в подробности). Следует ознакомиться с руководствами по тем подсистемам хранения данных, которые вы решите использовать.

Вкратце рассмотрим четыре уровня изолированности.

- `READ UNCOMMITTED`. На этом уровне изолированности транзакции могут видеть результаты незавершенных транзакций. Вы можете столкнуться с множеством проблем, если не знаете абсолютно точно, что делаете. Используйте этот уровень, только если у вас есть на то веские причины. На практике этот уровень применяется редко, поскольку в этом случае производительность лишь немного выше, чем на других уровнях, имеющих множество преимуществ. Чтение незавершенных данных называют еще *черновым*, или *«грязным» чтением* (dirty read).
- `READ COMMITTED`. Это уровень изолированности, который устанавливается по умолчанию в большинстве СУБД (но не в MySQL!). Он соответствует приведенному ранее простому определению изолированности: транзакция увидит только те изменения, которые к моменту ее начала подтверждены другими транзакциями, а произведенные ею изменения останутся невидимыми для других транзакций, пока текущая не будет подтверждена. На этом уровне возможно так называемое *неповторяющееся чтение* (nonrepeatable read). Это означает, что вы можете выполнить одну и ту же команду дважды и получить разный результат.
- `REPEATABLE READ`. Этот уровень изолированности позволяет решить проблемы, которые возникают на уровне `READ UNCOMMITTED`. Он гарантирует, что любые строки, которые считываются транзакцией, будут выглядеть одинаково при последовательных операциях чтения в пределах одной транзакции, однако теоретически на этом уровне возможна другая проблема, которая называется *фантомным чтением* (phantom reads). Проще говоря, фантомное чтение может произойти в случае, если вы выбираете некоторый диапазон строк, затем другая транзакция вставляет в него новую строку, после чего вы снова выбираете тот же диапазон. В результате вы увидите новую, фантомную строку. InnoDB и XtraDB решают проблему фантомного чтения с помощью многоверсионного управления конкурентным доступом (multiversion concurrency control), о котором мы расскажем далее в этой главе.

Уровень изолированности `REPEATABLE READ` устанавливается в MySQL по умолчанию.

- `SERIALIZABLE`. Самый высокий уровень изолированности, который решает проблему фантомного чтения, заставляя транзакции выполняться в таком порядке, чтобы исключить возможность конфликта. Если коротко, уровень `SERIALIZABLE` блокирует каждую читаемую строку. На этом уровне может возникать множество задержек и конфликтов блокировок. Нам редко встречались люди, использующие этот уровень, но потребности вашего приложения могут заставить применять его, смирившись с меньшей степенью конкурентного доступа, но обеспечивая стабильность данных.

В табл. 1.1 приведена сводка различных уровней изолированности и указаны недостатки, присущие каждому из них.

Таблица 1.1. Уровни изолированности ANSI SQL

Уровень изоляции	Возможность чернового чтения	Возможность неповторяющегося чтения	Возможность фантомного чтения	Блокировка чтения
READ UNCOMMITTED	Да	Да	Да	Нет
READ COMMITTED	Нет	Да	Да	Нет
REPEATABLE READ	Нет	Нет	Да	Нет
SERIALIZABLE	Нет	Нет	Нет	Да

Взаимоблокировки

Взаимоблокировка возникает тогда, когда две и более транзакции взаимно удерживают и запрашивают блокировку одних и тех же ресурсов, создавая циклическую зависимость. Такие состояния наблюдаются и в том случае, если транзакции пытаются заблокировать ресурсы в разном порядке. Они могут возникнуть, когда несколько транзакций блокируют одни и те же ресурсы. Для примера рассмотрим две транзакции, обращающиеся к таблице `StockPrice`:

Транзакция № 1

```
START TRANSACTION;
UPDATE StockPrice SET close = 45.50 WHERE stock_id = 4 and date = '2002-05-01';
UPDATE StockPrice SET close = 19.80 WHERE stock_id = 3 and date = '2002-05-02';
COMMIT;
```

Транзакция № 2

```
START TRANSACTION;
UPDATE StockPrice SET high = 20.12 WHERE stock_id = 3 and date = '2002-05-02';
UPDATE StockPrice SET high = 47.20 WHERE stock_id = 4 and date = '2002-05-01';
COMMIT;
```

Если вам не повезет, то каждая транзакция выполнит свой первый запрос и обновит строку данных, заблокировав ее. Затем все транзакции попытаются обновить вторую строку, но обнаружат, что та уже заблокирована. В итоге каждая транзакция будет до бесконечности ожидать окончания другой, пока не произойдет вмешательство извне, которое снимет взаимоблокировку.

Для борьбы с этой проблемой в СУБД реализованы различные формы обнаружения взаимоблокировок и тайм-аутов. Более совершенные подсистемы хранения данных, такие как InnoDB, легко обнаруживают циклические зависимости и немедленно возвращают ошибку. Это очень хорошо, иначе взаимоблокировки проявлялись бы в виде очень медленных запросов. Другие системы откатывают транзакцию по истечении тайм-аута, что не очень хорошо. InnoDB обрабатывает взаимоблокировки откатом той транзакции, которая захватила меньше всего монопольных блокировок строк (приблизительный показатель легкости отката).

Поведение и порядок блокировок зависят от подсистемы хранения данных, так что в одних подсистемах при определенной последовательности команд могут происходить взаимоблокировки, а в других — нет. Взаимоблокировки имеют двойственную природу: некоторые неизбежны из-за конфликта данных, другие вызваны схемой работы конкретной подсистемы хранения.

Нельзя справиться с взаимоблокировками без отката одной из транзакций, частичного либо полного. Такова суровая правда жизни в транзакционных системах, и это надо учитывать при проектировании приложений. Многие приложения могут просто попытаться выполнить транзакцию с самого начала.

Ведение журнала транзакций

Ведение журнала помогает сделать транзакции более эффективными. Вместо обновления таблиц на диске после каждого изменения подсистема хранения данных может изменить находящуюся в памяти копию данных. Это происходит очень быстро. Затем подсистема хранения запишет сведения об изменениях в журнал транзакции, который хранится на диске и поэтому долговечен. Это тоже довольно быстрая операция, поскольку добавление событий в журнал сводится к операции последовательного ввода/вывода в пределах ограниченной области диска вместо случайного ввода/вывода в разных местах. Позже процесс обновит таблицу на диске. Таким образом, большинство подсистем хранения данных, которые используют этот метод *(упреждающую запись в журнал)*, дважды сохраняют изменения на диске.

Если сбой произойдет после внесения записи в журнал транзакции, но до обновления самих данных, подсистема хранения может восстановить изменения после перезагрузки сервера. Методы восстановления у каждой подсистемы хранения данных различны.

Транзакции с MySQL

MySQL предоставляет пользователям две транзакционные подсистемы хранения данных: InnoDB и NDB Cluster. Существует также несколько подсистем сторонних разработчиков. Наиболее известны сейчас XtraDB и PBXT. В следующем разделе мы обсудим некоторые свойства каждой из них.

AUTOCOMMIT

По умолчанию MySQL работает в режиме `AUTOCOMMIT`. Это означает, что, пока вы не начали транзакцию явно, каждый запрос автоматически выполняется в отдельной транзакции. Вы можете установить или отключить режим `AUTOCOMMIT` для текущего соединения, задав значение переменной:

```
mysql> SHOW VARIABLES LIKE 'AUTOCOMMIT';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.00 sec)
mysql> SET AUTOCOMMIT = 1;
```

Значения `1` и `ON` эквивалентны, так же как `0` и `OFF`. После отправки запроса в режиме `AUTOCOMMIT=0` вы оказываетесь в транзакции, пока не выполните команду `COMMIT` или `ROLLBACK`. После этого MySQL немедленно начинает новую транзакцию. Изменение значения переменной `AUTOCOMMIT` не влияет на нетранзакционные таблицы, такие как `MyISAM` или `Memory`, которые не имеют понятия о подтверждении или отмене транзакций.

Некоторые команды, будучи запущенными во время начатой транзакции, заставляют MySQL подтвердить транзакцию до их выполнения. Обычно это команды языка определения данных (Data Definition Language, DDL), которые вносят изменения в структуру таблиц, например `ALTER TABLE`, но `LOCK TABLES` и другие директивы также обладают этим свойством. В документации к своей версии MySQL вы можете найти полный список команд, автоматически фиксирующих транзакцию.

MySQL позволяет устанавливать уровень изолированности с помощью команды `SET TRANSACTION ISOLATION LEVEL`, которая начинает действовать со следующей транзакции. Можете настроить уровень изолированности для всего сервера в конфигурационном файле или только для своей сессии:

```
mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

MySQL распознает все четыре стандартных уровня изоляции ANSI, а InnoDB все их поддерживает.

Использование нескольких подсистем хранения данных в транзакциях

MySQL не управляет транзакциями на уровне сервера. Вместо этого подсистемы хранения данных реализуют транзакции самостоятельно. Это означает, что вы не можете надежно сочетать различные подсистемы в одной транзакции.

Если вы используете транзакционные и нетранзакционные таблицы (например, таблицы InnoDB и MyISAM) в одной транзакции, то все будет работать хорошо, пока не произойдет что-то неожиданное.

Однако если потребуется выполнить откат, то невозможно будет отменить изменения, внесенные в нетранзакционную таблицу. Из-за этого база данных становится несогласованной, и восстановить ее после такого события нелегко, что ставит под сомнение идею транзакций в целом. Вот почему так важно выбирать для каждой таблицы подходящую подсистему хранения.

MySQL обычно не предупреждает и не выдает сообщений об ошибках, если вы выполняете транзакционные операции над нетранзакционной таблицей. Иногда при откате транзакции может быть сгенерировано предупреждение **Some nontransactional changed tables couldn't be rolled back** (Откат некоторых измененных нетранзакционных таблиц невозможен), но большую часть времени вы не будете знать о том, что работаете с нетранзакционными таблицами.

Явные и неявные блокировки

В подсистеме хранения InnoDB применяется двухфазный протокол блокировки. Она может устанавливать блокировки в любой момент транзакции, но не снимает их до выполнения команд `COMMIT` или `ROLLBACK`. Все блокировки снимаются одновременно. Ранее описанные механизмы блокировки являются неявными. InnoDB обрабатывает блокировки автоматически в соответствии с вашим уровнем изоляции.

Однако InnoDB поддерживает и явную блокировку, которая в стандарте SQL вообще не упоминается[1]:

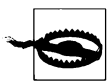
- `SELECT ... LOCK IN SHARE MODE;`
- `SELECT ... FOR UPDATE.`

MySQL также поддерживает команды `LOCK TABLES` и `UNLOCK TABLES`, которые реализуются на сервере, а не в подсистеме хранения. Они применяются в определенных случаях, но не служат заменой транзакциям. Если вам нужны транзакции, используйте транзакционную подсистему хранения.

Нам часто попадаются приложения, которые были перенесены из MyISAM в InnoDB, но в которых по-прежнему используется команда `LOCK TABLES`. В этой команде боль-

[1] Блокирующими подсказками зачастую злоупотребляют, поэтому их стоит избегать. Подробнее об этом говорится в главе 6.

ше нет необходимости, так как применяются построчные блокировки, а проблемы с производительностью она может вызывать серьезные.



Команда LOCK TABLES плохо взаимодействует с транзакциями, и в некоторых версиях сервера и те и другие ведут себя непредсказуемо. Поэтому мы рекомендуем применять команду LOCK TABLES только в рамках транзакции с режимом AUTOCOMMIT независимо от того, какой подсистемой хранения вы пользуетесь.

Управление конкурентным доступом с помощью многоверсионности

Большинство транзакционных подсистем хранения в MySQL не используют простой механизм построчной блокировки. Они применяют построчную блокировку в сочетании с методикой увеличения конкурентности, известной как *управление конкурентным доступом с помощью многоверсионности* (multiversion concurrency control, MVCC). Нельзя сказать, что методика MVCC присуща исключительно MySQL — она используется также в Oracle, PostgreSQL и некоторых других СУБД, хотя из-за отсутствия стандарта возможны существенные различия в их работе.

Вы можете воспринимать MVCC как вариацию построчной блокировки. Во многих случаях оно позволяет обойтись без блокировки и существенно снизить издержки. В зависимости от способа реализации MVCC может допускать чтение без блокировки, блокируя только необходимые строки во время операций записи.

MVCC сохраняет мгновенный снимок состояния данных в определенный момент времени. Это означает, что транзакции вне зависимости от их длительности могут видеть согласованные данные. А также что различные транзакции могут видеть разные данные в одних и тех же таблицах в одно и то же время! Если вы никогда не сталкивались с этим раньше, то можете сильно удивиться, но по мере знакомства с этой технологией все становится понятнее.

Каждая подсистема хранения реализует MVCC по-своему. В некоторых вариантах применяется оптимистическое и пессимистическое управление конкурентным доступом. Мы проиллюстрируем один из способов работы MVCC, объяснив упрощенную версию поведения InnoDB.

InnoDB реализует MVCC, сохраняя вместе с каждой строкой два скрытых значения, которые показывают, когда строка была создана и когда истек срок ее хранения (или она была удалена). Вместо хранения фактического момента времени, когда произошли данные события, строка хранит номер версии системы для этого момента. В начале каждой транзакции этот номер увеличивается на единицу. Каждая транзакция хранит собственную запись текущей версии системы на момент своего начала. Каждый запрос должен сравнивать номера версий каждой строки с версией

транзакции. Посмотрим, как эта методика применяется к конкретным операциям, когда транзакция имеет уровень изоляции REPEATABLE READ.

- SELECT. InnoDB должна проверить каждую строку на соответствие двум критериям.
  - Найти версию строки, по крайней мере такой же старой, как версия транзакции (то есть ее номер версии должен быть меньше номера версии транзакции или равен ему). Это гарантирует, что либо строка существовала до начала транзакции, либо транзакция создала или изменила эту строку.
  - Версия удаления строки должна быть не определена, или ее значение должно быть больше, чем версия транзакции. Это гарантирует, что строка не была удалена до начала транзакции.

  Строки, прошедшие обе проверки, могут быть возвращены в качестве результата запроса.
- INSERT. InnoDB записывает текущий номер версии системы вместе с новой строкой.
- DELETE. InnoDB записывает текущий номер версии системы как ID удаления строки.
- UPDATE. InnoDB записывает новую копию строки, используя номер версии системы в качестве версии новой строки. Она также записывает номер версии системы как версию удаления старой строки.

Благодаря хранению дополнительных записей большинство запросов на чтение никогда не будут ставить блокировки. Они просто как можно быстрее считывают данные, выбирая только те строки, которые удовлетворяют заданному критерию. Недостатком такого подхода является то, что подсистема хранения должна записывать для каждой строки дополнительные данные, выполнять лишнюю работу при проверке строк и производить некоторые дополнительные служебные операции.

MVCC работает только на уровнях изолированности REPEATABLE READ и READ COMMITTED. Уровень READ UNCOMMITTED несовместим с MVCC[1], поскольку запросы не считывают версию строки, соответствующую их версии транзакции. Несмотря ни на что они читают самую последнюю версию. Уровень SERIALIZABLE несовместим с MVCC, поскольку операции чтения блокируют каждую возвращаемую строку.

Подсистемы хранения в MySQL

В этом разделе представлен обзор подсистем хранения MySQL. Мы не будем вдаваться в подробности, поскольку планируем обсуждать подсистемы хранения и особенности их работы на протяжении всей книги. Однако учитывайте, что это издание не является исчерпывающим источником информации. Вам следует изучить документацию MySQL к выбранной подсистеме хранения.

[1] Официального стандарта, определяющего MVCC, не существует, поэтому разные подсистемы хранения и СУБД реализуют его по-разному, и нельзя сказать, что какой-то вариант является ошибочным.

MySQL хранит каждую базу данных (также именуемую *схемой*) как подкаталог своего каталога данных в файловой системе. Когда вы создаете таблицу, MySQL сохраняет ее определение в файле с расширением `.frm` и именем, совпадающим с именем таблицы. Таким образом, при создании таблицы с именем `MyTable` ее определение сохраняется в файле `MyTable.frm`. Поскольку MySQL использует файловую систему для хранения имен баз данных и определений таблиц, чувствительность к регистру символов зависит от платформы. В MySQL для Windows имена таблиц и баз данных нечувствительны к регистру, а в операционных системах семейства UNIX — чувствительны. Каждая подсистема хранения по-разному записывает табличные данные и индексы, но сервер сам обрабатывает определение таблицы.

Для получения информации о таблицах можете использовать команду `SHOW TABLE STATUS` (или, начиная с версии MySQL 5.0, сделать запрос таблиц `INFORMATION_SCHEMA`). Например, чтобы получить информацию о таблице `user`, содержащейся в базе данных `mysql`, выполните следующую команду:

```
mysql> SHOW TABLE STATUS LIKE 'user' \G
*************************** 1. row ***************************
           Name: user
         Engine: MyISAM
      Row_format: Dynamic
           Rows: 6
 Avg_row_length: 59
    Data_length: 356
Max_data_length: 4294967295
   Index_length: 2048
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2002-01-24 18:07:17
    Update_time: 2002-01-24 21:56:29
     Check_time: NULL
      Collation: utf8_bin
       Checksum: NULL
 Create_options:
        Comment: Users and global privileges
1 row in set (0.00 sec)
```

Как видите, это таблица типа MyISAM. Команда выдала также много дополнительной информации и статистики. Давайте вкратце рассмотрим, что означает каждая строка:

- `Name` — имя таблицы;
- `Engine` — подсистема хранения. В старых версиях MySQL этот столбец назывался `Type`, а не `Engine`;
- `Row_format` — формат строки. Для таблицы MyISAM он может иметь значение `Dynamic`, `Fixed` или `Compressed`. Длина динамических строк может меняться, поскольку они содержат поля переменной длины типа `VARCHAR` или `BLOB`. Фиксированные строки имеют один и тот же размер и состоят из полей постоянной длины, таких как `CHAR` и `INTEGER`. Сжатые строки существуют только в сжатых таблицах (см. далее подраздел «Сжатые таблицы MyISAM»);

- `Rows` — количество строк в таблице. Для таблиц MyISAM и большинства других подсистем хранения это число всегда точное, для InnoDB — обычно приблизительное;
- `Avg_row_length` — количество байтов (в среднем), содержащееся в каждой строке;
- `Data_length` — объем данных (в байтах) во всей таблице;
- `Max_data_length` — максимальный объем данных, который может хранить эта таблица (зависит от подсистемы хранения);
- `Index_length` — объем дискового пространства, занятый индексными данными;
- `Data_free` — для таблицы MyISAM показывает объем выделенного пространства, которое в данный момент не используется. Это пространство служит для хранения ранее удаленных строк и может быть задействовано в будущем при выполнении команд `INSERT`;
- `Auto_increment` — следующее значение атрибута `AUTO_INCREMENT`;
- `Create_time` — момент создания таблицы;
- `Update_time` — время последнего изменения таблицы;
- `Check_time` — время последней проверки таблицы командой `CHECK TABLE` или утилитой myisamchk;
- `Collation` — устанавливаемая по умолчанию кодировка и схема упорядочения для символьных столбцов в этой таблице;
- `Checksum` — текущая контрольная сумма содержимого всей таблицы, если ее можно определить;
- `Create_options` — любые другие параметры, которые были указаны при создании таблицы;
- `Comment` — это поле содержит различную дополнительную информацию. Для таблиц MyISAM в нем хранятся комментарии, добавленные при их создании. Если таблица использует подсистему хранения InnoDB, здесь указан объем свободного места в табличном пространстве. Для представлений комментарий содержит текст `VIEW`.

Подсистема хранения InnoDB

InnoDB является транзакционной подсистемой хранения по умолчанию в MySQL, а также наиболее значимой и широко используемой подсистемой хранения в целом. Она была создана для обработки большого количества краткосрочных транзакций, которые выполняются благополучно намного чаще, чем откатываются. Высокая производительность и автоматическое восстановление после сбоя делают ее популярной и для нетранзакционных целей. *Вам следует применять InnoDB для своих таблиц, пока не возникнет необходимость использовать другую подсистему хранения.* Если вы хотите изучить подсистемы хранения, не стоит рассматривать их все слишком подробно, но обязательно потратьте время на глубокое ознакомление с InnoDB, чтобы узнать о ней как можно больше.

История InnoDB

История релизов InnoDB довольно сильно запутана, но очень помогает разобраться в этой подсистеме хранения данных. В 2008 году для версии MySQL 5.1 был выпущен так называемый плагин InnoDB. Это было следующее поколение InnoDB, созданное компанией Oracle, которой в то время принадлежала InnoDB, но не MySQL. По разным причинам, которые лучше обсуждать за кружкой пива, MySQL продолжала поставлять более старую версию InnoDB, скомпилированную на сервер. Но вы могли по собственному желанию отключить ее и установить новый, более эффективный и лучше масштабируемый плагин InnoDB. В конце концов компания Oracle приобрела компанию Sun Microsystems и, следовательно, СУБД MySQL и удалила старую кодовую базу, заменив ее «плагином» по умолчанию в версии MySQL 5.5. (Да, это означает, что теперь «плагин» фактически скомпилирован, а не установлен как плагин. Старая терминология изживается с трудом.)

Современная версия InnoDB, представленная в качестве плагина InnoDB в MySQL 5.1, обеспечивает новый функционал, например построение индексов путем сортировки, возможность удаления и добавления индексов без перестройки всей таблицы, новый формат хранения данных, который предполагает сжатие, новый способ хранения больших объемов данных, таких как столбцы `BLOB`, и управления форматом файлов. Многие люди, которые работают с MySQL 5.1, не применяют этот плагин, чаще всего потому, что не подозревают о нем. Если вы используете MySQL 5.1, убедитесь, пожалуйста, в том, что применяете плагин InnoDB. Он намного лучше более ранней версии InnoDB.

InnoDB настолько важна, что в ее разработку внесли свой вклад не только команда Oracle, но и многие другие люди и компании, в частности Ясуфуми Киносита (Yasufumi Kinoshita), а также компании Google, Percona и Facebook. Некоторые из внесенных ими усовершенствований были включены в официальный исходный код InnoDB, многие другие были немного переработаны командой InnoDB и затем внедрены. В целом развитие InnoDB значительно ускорилось за последние несколько лет, улучшения коснулись инструментария, масштабируемости, способности к изменению конфигурации, производительности, функций и поддержки для Windows и прочих важных вещей. Лабораторные превью и релизы ключевых изменений, вносимых в версию MySQL 5.6, также представляют множество замечательных новых функций InnoDB.

Oracle инвестирует колоссальные ресурсы и проделывает огромную работу для улучшения производительности InnoDB (и здесь очень полезным оказывается вклад, который вносят внешние разработчики). Во втором издании этой книги мы отмечали, что InnoDB выглядела довольно жалко, работая на основе четырехпроцессорных ядер. Теперь она хорошо масштабируется до 24 ядер процессора, а возможно, и до 32 или даже большего количества в зависимости от сценария.

Обзор InnoDB

InnoDB сохраняет данные в одном или нескольких файлах данных, которые называются *табличным пространством* (tablespace). Табличное пространство, в сущности, является черным ящиком, которым управляет сама InnoDB. В MySQL 4.1 и более поздних версиях InnoDB может хранить данные и индексы каждой таблицы в отдельных файлах. Кроме того, она может располагать табличные пространства в «сырых» (неформатированных) разделах диска. Но современные файловые системы делают эту возможность бессмысленной.

InnoDB использует MVCC для обеспечения высокой степени конкурентности и реализует все четыре стандартных уровня изолированности SQL. Уровнем изоляции по умолчанию является **REPEATABLE READ**, а стратегия *блокировки следующего ключа* предотвращает фантомные чтения на этом уровне: вместо того чтобы блокировать только строки, затронутые в запросе, InnoDB блокирует пропуски в структуре индекса, предотвращая вставку фантомных строк.

Таблицы InnoDB строятся на *кластеризованных индексах*, которые мы детально рассмотрим в следующих главах. Структуры индексов в InnoDB значительно отличаются от используемых в других подсистемах хранения. В результате эта подсистема обеспечивает более быстрый поиск по первичному ключу. Однако *вторичные индексы* (индексы, не являющиеся первичным ключом) содержат все столбцы первичного ключа, так что если первичный ключ большой, то все прочие индексы тоже будут большими. Если в таблице планируется много индексов, нужно стремиться к тому, чтобы первичный ключ был небольшим. Формат хранения данных не зависит от платформы. Это означает, что вы можете без проблем скопировать файлы данных и индексов с сервера Intel на PowerPC или Sun SPARCI.

InnoDB поддерживает множество внутренних оптимизаций. В их число входят прогнозное упреждающее чтение для предварительной выборки данных с диска, адаптивный хеш-индекс, который автоматически выстраивает хеш-индексы в памяти для обеспечения очень быстрого поиска, и буфер вставок для ускорения операций вставки. Мы еще рассмотрим эти вопросы.

Подсистема InnoDB очень сложна, и если вы ее используете, то мы настоятельно рекомендуем вам прочитать раздел «InnoDB Transaction Model and Locking» («Транзакционная модель и блокировки в InnoDB») руководства по MySQL. Из-за наличия архитектуры MVCC в работе подсистемы InnoDB есть много тонкостей, о которых вы должны узнать прежде, чем создавать приложения. Работа с подсистемой хранения, поддерживающей согласованные представления данных для всех пользователей, даже когда некоторые пользователи меняют данные, может быть сложной.

В качестве транзакционной подсистемы хранения InnoDB поддерживает горячее онлайновое резервное копирование (см. главу 15) с помощью различных механизмов, включая запатентованную Oracle Enterprise Backup и Percona XtraBackup с открытым исходным кодом. В других подсистемах хранения MySQL нет горячих резервных копий — чтобы получить согласованную резервную копию, вам необходимо остановить все процессы записи в таблицу, которые при смешанной рабочей нагрузке на чтение и запись обычно заканчиваются также остановкой чтения.

Подсистема хранения MyISAM

Будучи подсистемой хранения по умолчанию в MySQL 5.1 и более ранних версиях, MyISAM предоставляет большой список функций, таких как полнотекстовое индексирование, сжатие и пространственные функции (для геоинформационных систем, ГИС). MyISAM не поддерживает транзакции или построчные блокировки. Ее самым слабым местом, несомненно, является то, что она не имеет даже удаленного механизма защиты от сбоев. Из-за подсистемы MyISAM MySQL до сих пор имеет репутацию СУБД без транзакций, хотя позволяет использовать транзакции уже более десяти лет! Тем не менее MyISAM не так уж плоха для нетранзакционной, не отказоустойчивой подсистемы хранения. Если вам нужны данные только для чтения или если ваши таблицы невелики и их восстановление не будет чересчур сложным, то не должно возникнуть вопросов по использованию MyISAM. (Но, пожалуйста, не применяйте ее по умолчанию. Вместо этого задействуйте InnoDB.)

Хранение

MyISAM обычно хранит каждую таблицу в двух файлах — в файле данных и индексном файле. Эти файлы имеют расширения `.MYD` и `.MYI` соответственно. Таблицы типа MyISAM могут содержать как динамические, так и статические строки (строки фиксированной длины). MySQL решает, какой формат использовать, основываясь на определении таблицы. Количество строк в таблице типа MyISAM ограничено в первую очередь доступным дисковым пространством на сервере базы данных и максимальным размером файла, допустимым в операционной системе.

Таблицы MyISAM со строками переменной длины, создаваемые в версии MySQL 5.0, по умолчанию настроены на поддержку 256 Тбайт данных с использованием шестибайтных указателей на записи с данными. В более ранних версиях MySQL указатели по умолчанию были четырехбайтными с максимальным объемом данных 4 Гбайт. Все версии MySQL могут поддерживать размер указателя до 8 байт. Чтобы изменить размер указателя в таблице MyISAM (уменьшить или увеличить), вы должны изменить таблицу и задать новые значения параметров `MAX_ROWS` и `AVG_ROW_LENGTH`, которые дают приблизительную оценку необходимого пространства. Это приведет к перезаписи таблицы и всех ее индексов, что может занять много времени.

Особенности MyISAM

Как одна из самых старых подсистем хранения MySQL, MyISAM может выполнять много функций, которые за годы использования СУБД были разработаны для решения различных задач.

- *Блокирование и конкурентный доступ.* MyISAM может блокировать только таблицы целиком, не построчно. Запросы на чтение получают разделяемые (на чтение) блокировки всех таблиц, которые им нужно прочитать. Запросы на запись получают монопольные (на запись) блокировки. Однако вы можете вставлять новые строки в таблицу в то время, когда исполняются запросы на выборку данных из таблицы (конкурентные вставки).

- *Исправление данных.* MySQL поддерживает ручные и автоматические проверку и исправление таблиц типа MyISAM, но не стоит путать эти функции с транзакциями или аварийным восстановлением. После исправления таблицы вы, скорее всего, обнаружите, что некоторые данные просто исчезли. К тому же исправление работает очень медленно. Вы можете применять команды `CHECK TABLE mytable` и `REPAIR TABLE mytable` для проверки таблицы на предмет наличия ошибок и их устранения. А также использовать командную утилиту myisamchk для проверки и исправления таблиц, когда сервер находится в автономном (offline) режиме.
- *Особенности индексирования.* Вы можете создавать индексы по первым 500 символам столбцов типа `BLOB` и `TEXT` в таблицах MyISAM. MyISAM поддерживает полнотекстовые индексы, которые индексируют отдельные слова для сложных операций поиска. Дополнительную информацию об индексировании можно найти в главе 5.
- *Отложенная запись ключей.* Таблицы MyISAM, созданные с пометкой `DELAY_KEY_WRITE`, не записывают измененные индексы на диск в конце запроса. Вместо этого MyISAM сохраняет изменения в буфере памяти для ключей. Сброс индексных блоков на диск происходит при переполнении буфера или закрытии таблицы. Это позволяет увеличить производительность, но в случае сбоя сервера или системы индексы наверняка будут повреждены и потребуют восстановления. Вы можете настроить отложенную запись ключей как для всей базы, так и для отдельных таблиц.

Сжатые таблицы MyISAM

Некоторые таблицы после создания и заполнения данными никогда больше не изменяются. Такие таблицы хорошо подходят для сжатия средствами MyISAM.

Для сжатия (или упаковки) таблиц существует специальная утилита myisampack. Сжатые таблицы невозможно изменить (хотя при необходимости можно распаковать их, внести изменения и снова сжать), но они занимают на диске гораздо меньше места, чем неупакованные. В результате их использования увеличивается производительность, поскольку из-за небольшого размера таких таблиц требуется меньше операций поиска на диске для нахождения требуемых записей. Сжатые таблицы MyISAM могут иметь индексы, но они также доступны только для чтения.

Издержки на распаковку данных для чтения незначительны для большинства приложений, установленных на современном оборудовании, а значительная экономия достигается из-за сокращения операций ввода/вывода. Строки сжимаются по отдельности, поэтому для получения одной строки MySQL не нужно распаковывать всю таблицу (или даже страницу).

Производительность MyISAM

Благодаря компактному хранению данных и невысоким издержкам, обусловленным простотой архитектуры, MyISAM в некоторых случаях может обеспечить хорошую производительность. У нее есть серьезные ограничения по масштабируемости, включая мьютексы, установленные на ключевых кэшах. MariaDB предлагает сег-

ментированный кэш-ключ, который позволяет избежать этой проблемы. Однако наиболее распространенной проблемой MyISAM, мешающей добиться высокой производительности, является блокировка таблиц. Если запросы застревают в статусе «Заблокировано», вы страдаете от блокировки на уровне таблицы.

Другие встроенные подсистемы хранения данных MySQL

MySQL предоставляет множество специализированных подсистем хранения. По разным причинам большинство из них несколько устарели для новых версий. Некоторые же по-прежнему доступны на сервере, но требуют специальной активации.

Archive

Подсистема хранения Archive позволяет выполнять только команды `INSERT` и `SELECT`, к тому же до версии MySQL 5.1 она не поддерживала индексирование. Archive требует значительно меньше операций дискового ввода/вывода, чем MyISAM, поскольку буферизует записываемые данные и сжимает все вставляемые строки с помощью библиотеки zlib. Кроме того, каждый запрос `SELECT` требует полного просмотра таблицы. Таблицы Archive идеальны для ведения журнала и сбора данных, когда анализ чаще всего сводится к просмотру всей таблицы или когда требуется обеспечить быстроту выполнения запросов `INSERT`.

Подсистема Archive поддерживает построчную блокировку и специальную буферную систему для вставки с высокой степенью конкурентности. Она обеспечивает согласованное чтение, останавливая выполнение команды `SELECT` после извлечения того количества строк, которое было в таблице к началу выполнения запроса. Она также обеспечивает невидимость результатов пакетной вставки до завершения процедуры. Эти особенности эмулируют некоторые виды поведения транзакций и MVCC, но Archive не является транзакционной подсистемой хранения. Она лишь оптимизирована для высокоскоростной вставки и хранения данных в сжатом виде.

Blackhole

В подсистеме Blackhole механизма хранения данных вообще нет. Она игнорирует команды `INSERT`, а не хранит их. Однако сервер записывает запросы к таблицам Blackhole в журналы как обычно, поэтому их можно реплицировать на подчиненные серверы или просто сохранить в журнале. Это делает подсистему Blackhole полезной для настройки предполагаемых репликаций и ведения журнала аудита, но мы видели много проблем, вызванных такими настройками, и не рекомендуем применять этот способ.

CSV

Подсистема хранения CSV может обрабатывать файлы с разделителями запятыми (comma-separated values, CSV) как таблицы, но не поддерживает индексы по ним. Эта подсистема позволяет импортировать и экспортировать данные во время работы

сервера. Если вы экспортируете CSV-файл из электронной таблицы и сохраните в каталоге данных сервера MySQL, то сервер сможет сразу его прочитать. Аналогично, если вы записываете данные в таблицу CSV, внешняя программа сможет сразу же ее прочесть. Таким образом, таблицы CSV полезны в качестве формата обмена данными.

Federated

Эта подсистема хранения является своего рода прокси-сервером для других серверов. Она открывает клиентское соединение с другим сервером и выполняет там запросы к таблице, получая и отправляя строки по мере необходимости. Первоначально она продавалась в качестве конкурента для функций, поддерживаемых многими частными серверами баз данных корпоративного уровня, такими как Microsoft SQL Server и Oracle, но это было, мягко говоря, притянуто за уши. Подсистема хранения Federated выглядит способной обеспечивать высокую степень гибкости, однако она является источником многих проблем и по умолчанию отключена. Ее преемница FederatedX доступна в MariaDB.

Memory

Таблицы типа Memory (раньше называвшиеся таблицами типа HEAP) удобно использовать, когда необходимо получить быстрый доступ к данным, которые либо никогда не изменяются, либо не должны сохраняться после перезапуска сервера. Таблицы типа Memory обрабатываются на порядок быстрее, чем таблицы MyISAM. Все содержащиеся в них данные хранятся в памяти, поэтому запросам не нужно ждать выполнения операций дискового ввода/вывода. Структура таблицы Memory сохраняется после перезагрузки сервера, но данные теряются.

Вот несколько способов применения таблиц типа Memory:

- в качестве справочных таблиц или таблиц соответствия, таких как таблица, в которой почтовым индексам соответствуют названия регионов;
- для кэширования результатов периодического агрегирования данных;
- для получения промежуточных результатов при анализе данных.

Таблицы Memory поддерживают HASH-индексы, которые обеспечивают очень высокую скорость выполнения поисковых запросов. Эти таблицы работают очень быстро, но не всегда подходят для замены дисковых таблиц. Они используют табличную блокировку, что уменьшает конкуренцию при записи. Они не поддерживают столбцы типа `TEXT` и `BLOB` и допускают использование только строк фиксированной длины, поэтому значения типа `VARCHAR` сохраняются как значения типа `CHAR`, что увеличивает расход памяти. (Некоторые из этих ограничений сняты в Percona Server.)

MySQL использует подсистему Memory для внутреннего хранения промежуточных результатов при обработке запросов, которым требуется временная таблица. Если промежуточный результат становится слишком большим для таблицы Memory или

содержит столбцы типа TEXT или BLOB, то MySQL преобразует его в таблицу MyISAM на диске. В следующих главах мы обсудим это подробнее.

Многие путают таблицы типа Memory с временными таблицами, которые создаются командой CREATE TEMPORARY TABLE. Временные таблицы могут использовать любую подсистему хранения. Это не то же самое, что таблицы типа Memory. Временные таблицы видны только в одном соединении и исчезают, когда оно закрывается.

Merge

Подсистема хранения Merge является вариацией подсистемы MyISAM. Таблица типа Merge представляет собой виртуальную таблицу, состоящую из нескольких одинаковых по структуре таблиц MyISAM. Это может быть полезно, когда вы используете MySQL для ведения журнала и организации хранилища данных. Правда, эта возможность устарела — сейчас применяется секционирование (см. главу 7).

NDB Cluster

MySQL AB приобрела базу данных NDB у Sony Ericsson в 2003 году и создала подсистему хранения NDB Cluster в качестве интерфейса между SQL, используемым в MySQL, и собственным протоколом NDB. Комбинация сервера MySQL, подсистемы хранения NDB Cluster и распределенной, не поддерживающей разделения ресурсов, устойчивой к сбоям широкодоступной базы данных NDB называется MySQL Cluster. Мы еще обсудим MySQL Cluster позже.

Подсистемы хранения сторонних разработчиков

Поскольку MySQL предоставляет API для подключаемых подсистем хранения, с 2007 года появилось великое множество подсистем хранения, предназначенных для разных целей. Некоторые из них устанавливались на сервер, но большинство были продуктами сторонних разработчиков или проектами с открытым исходным кодом. Здесь мы обсудим часть подсистем хранения, которые, по нашему мнению, являются довольно полезными и все еще остаются актуальными.

Подсистемы хранения OLTP

Разработанная компанией Percona подсистема хранения XtraDB, интегрированная в Percona Server и MariaDB, является модифицированной версией InnoDB. Ее усовершенствования направлены на увеличение производительности, масштабируемости и операционной гибкости. Она может заменить InnoDB, так как способна читать

и записывать файлы данных, совместимые с InnoDB, и выполнять все запросы, которые может выполнять InnoDB.

Существует несколько других подсистем хранения OLTP, похожих на InnoDB в ключевых вопросах, например в обеспечении соответствия ACID и MVCC. Одной из них является подсистема PBXT, разработанная Полом Маккуллахом (Paul McCullagh) и компанией Primebase GMBH. В ней совмещаются репликация на уровне подсистемы, ограничения внешнего ключа и сложная архитектура, что делает ее пригодной для хранения на твердотельных накопителях и позволяет ей эффективно управлять большими значениями, например типа `BLOB`. PBXT широко применяется как подсистема совместного хранения и входит в состав MariaDB.

TokuDB использует новую структуру индексов, называемую fractal trees (фрактальные индексные деревья), которая не привязана к кэшу. По этой причине индексы не замедляют работу, когда их размер превышает объем памяти, не устаревают и не фрагментируются. TokuDB позиционируется на рынке как подсистема хранения больших данных, поскольку имеет высокие коэффициенты сжатия и может поддерживать множество индексов на больших объемах данных. На момент написания книги TokuDB представляет собой ранний релиз и имеет существенные ограничения конкурентности. Эти особенности делают подсистему отличным вариантом для работы с аналитическими наборами данных с интенсивной записью новых данных, но все может измениться в следующих версиях.

RethinkDB изначально позиционировалась как подсистема хранения, предназначенная для твердотельных накопителей, хотя, похоже, с течением времени она стала менее узкоспециализированной. Ее отличительной особенностью можно назвать применение работающей только на запись, копирующей только при записи структуры данных с использованием индексов, упорядоченных на основе B-деревьев. Эта подсистема все еще находится на ранней стадии разработки, и у нас не было возможности внимательно посмотреть на нее и оценить ее работу.

Подсистема Falcon продвигалась на рынок как следующее поколение транзакционной подсистемы хранения данных для MySQL в то время, когда компания Sun поглощала MySQL AB, но работа над ней давно прекращена. Джим Старки (Jim Starkey), главный архитектор Falcon, основал компанию для создания новой облачной NewSQL СУБД NuoDB (прежнее название NimbusDB).

Подсистемы хранения, ориентированные на столбцы

По умолчанию MySQL ориентируется на строки. Это означает, что данные каждой строки хранятся в одном месте и сервер при выполнении запросов ориентируется на строку как на рабочую единицу. Но при очень больших объемах данных более эффективным может быть ориентирование на столбцы. Это позволяет подсистеме извлекать меньше данных в ситуациях, когда строка целиком не нужна, а столбцы хранятся отдельно друг от друга, и их можно сжимать более качественно.

Ведущей подсистемой хранения, ориентированной на столбцы, является Infobright, которая хорошо работает с очень большими объемами данных (десятки терабайт).

Она спроектирована для использования в тех случаях, когда требуется аналитический подход или работа с хранилищем данных. Подсистема хранит данные в сильно сжатых блоках. А также хранит набор метаданных для каждого блока, что позволяет ей пропускать блоки или даже выполнять запросы, просто просматривая метаданные. Все дело в том, что Infobright не поддерживает индексы — при таких огромных размерах они бесполезны, а блочная структура представляет собой своего рода квазииндекс. Подсистема хранения требует пользовательских настроек сервера, потому что части сервера должны быть перезаписаны для работы с данными, ориентированными на столбцы. Некоторые запросы не могут быть выполнены подсистемой хранения в режиме, ориентированном на столбцы, поэтому сервер возвращается в режим, ориентированный на строки, что замедляет работу. Infobright существует как в версии с открытым исходным кодом, так и в коммерческой.

InfiniDB от Calpont — еще одна подсистема хранения данных, ориентированная на столбцы. Она доступна и в коммерческой версии, и в версии с открытым исходным кодом. InfiniDB предоставляет возможность делать запросы через кластер. Однако среди наших знакомых никто не использовал ее для работы.

Кстати, если вы хотите приобрести СУБД, ориентированную на столбцы, но не имеющую отношения к MySQL: мы оценили также LucidDB и MonetDB. Результаты эталонного тестирования и наше мнение о них вы найдете в MySQL Performance Blog, хотя с течением времени они, вероятно, несколько устареют.

Подсистемы хранения, создаваемые сообществом разработчиков

Полный список подсистем хранения данных от сообщества разработчиков насчитывал бы десятки или даже сотни систем, если бы мы принялись скрупулезно перечислять их. Однако можно с уверенностью сказать, что большинство из них обслуживают очень узкие ниши и практически неизвестны или используются малым количеством людей. Мы просто упомянем некоторые из них, так как большинство из них не видели в работе. *Caveat emptor!*[1]

- *Aria.* Aria, ранее называемая Maria, разрабатывается командой, создавшей MySQL, и позиционируется как преемница MyISAM. Она доступна в СУБД MariaDB. Многие из ее инструментов, которые изначально были заявлены, похоже, были отложены ради улучшения других частей сервера MariaDB. На момент написания книги, пожалуй, стоит охарактеризовать ее как защищенную от сбоев версию MyISAM, имеющую также несколько других усовершенствований, среди которых возможность кэшировать данные (а не только индексировать) в собственной памяти.
- *Groonga.* Это подсистема хранения данных с полнотекстовым поиском, которая, как утверждается, обеспечивает точность и высокую скорость.
- *OQGraph.* Эта подсистема хранения от Open Query поддерживает операции с графами (думаем, в виде «найдите кратчайший путь между узлами»), которые нецелесообразно или невозможно выполнять в SQL.

[1] Покупатель, будь бдителен! (Лат.)

- *Q4M*. Реализует очередь внутри MySQL с поддержкой операций, которые для самой SQL являются довольно сложными или невыполнимыми в рамках одной инструкции.
- *SphinxSE*. Эта подсистема хранения предоставляет интерфейс SQL для полнотекстового поискового сервера Sphinx, о котором мы подробнее поговорим в приложении E.
- *Spider*. Разделяет данные на несколько сегментов, эффективно реализуя прозрачное сегментирование, и выполняет запросы параллельно в различных сегментах данных, которые могут быть расположены на разных серверах.
- *VPForMySQL*. Эта подсистема хранения поддерживает вертикальное разбиение таблиц с помощью своего рода прокси-подсистемы хранения. То есть вы можете нарезать таблицу на несколько наборов столбцов и хранить их отдельно друг от друга, но использовать в запросах как одну таблицу. Разработчик тот же, что и у подсистемы Spider.

Выбор подходящей подсистемы хранения

Какую же подсистему хранения выбрать? InnoDB редко подводит, и мы очень рады, что компания Oracle сделала ее подсистемой по умолчанию в версии MySQL 5.5. Принятие решения о том, какую подсистему хранения применять, можно свести к фразе: «Используйте InnoDB, пока вам не понадобится какая-то функция, которую она не предоставляет и для которой нет хорошего альтернативного решения». Например, когда нужен полнотекстовый поиск, мы обычно используем InnoDB в сочетании со Sphinx, вместо того чтобы выбрать MyISAM с ее возможностью полнотекстового индексирования. Мы задействуем что-то отличное от InnoDB, когда нам не нужны функции InnoDB, а другая подсистема хранения обеспечивает существенные преимущества при отсутствии недостатков. Например, мы можем использовать MyISAM, когда нам не мешают ее ограниченная масштабируемость, слабая поддержка конкурентного доступа и отсутствие устойчивости к сбоям, а настоящей проблемой становится значительное потребление пространства подсистемой InnoDB.

Мы предпочитаем не смешивать и не комбинировать разные подсистемы хранения, если в этом нет острой необходимости. Такая практика существенно усложняет ситуацию, грозит проявлением множества потенциальных ошибок и имеет свои особенности. Взаимодействие между подсистемами хранения и сервером является довольно сложным и без внедрения туда же дополнительных подсистем хранения. Например, применение нескольких подсистем хранения мешает как следует настроить сервер или получить согласованные резервные копии.

Если вы считаете, что вам нужна другая подсистема хранения данных, то при выборе учитывайте следующие факторы.

- *Транзакции*. Если приложение нуждается в транзакциях, то наиболее стабильной, хорошо интегрированной, проверенной подсистемой хранения для него станет InnoDB (или XtraDB). MyISAM может стать хорошим вариантом, если задача

не требует транзакций и в основном предъявляет запросы типа `SELECT` или `INSERT`. Иногда в эту категорию попадают отдельные компоненты приложения, например ведение журнала.

- *Резервное копирование.* Необходимость регулярно выполнять резервное копирование также может повлиять на ваш выбор. Если есть возможность периодически останавливать работу сервера для выполнения этой процедуры, то подойдет любая подсистема хранения данных. Однако если требуется осуществлять резервное копирование без остановки сервера, то нужна InnoDB.
- *Восстановление после сбоя.* Если объем данных велик, то вы должны серьезно оценить количество времени, которое займет восстановление базы после сбоя. Таблицы MyISAM легче получают повреждения и требуют значительно больше времени для восстановления, чем таблицы InnoDB. Это одна из самых важных причин, по которой многие используют подсистему InnoDB, даже не нуждаясь в транзакциях.
- *Специальные возможности.* Наконец, может оказаться, что приложению требуются конкретные возможности или оптимизации, которые могут обеспечить лишь отдельные подсистемы хранения MySQL. Например, многие приложения используют оптимизацию кластерных индексов. В то же время только MyISAM поддерживает геопространственный поиск. Если подсистема хранения соответствует одному или нескольким важнейшим требованиям, но не соответствует другим, то нужно либо найти компромисс, либо выбрать разумное проектное решение. Часто вы можете получить то, что нужно, от подсистемы хранения, которая на первый взгляд не соответствует вашим требованиям.

Нет необходимости принимать решение прямо сейчас. Далее вы найдете в книге множество материалов, касающихся сильных и слабых сторон каждой подсистемы хранения, а также немало советов по архитектуре и проектированию. В общем, возможностей, вероятно, значительно больше, чем вы пока способны представить, а дальнейшее чтение поможет вам сделать оптимальный выбор. Если вы сомневаетесь, просто смотрите в сторону InnoDB. Она безопасна по умолчанию, и нет причин выбирать что-то иное, если вы еще четко не знаете, что вам нужно.

Все сказанное может показаться несколько абстрактным в отрыве от практики, так что давайте обратимся к некоторым широко распространенным приложениям баз данных. Мы рассмотрим различные таблицы и определим, какая подсистема хранения лучше всего удовлетворяет потребности каждой из них. Итоговую сводку вариантов приведем в следующем разделе.

Журналирование

Предположим, вы хотите использовать MySQL для ведения в режиме реального времени журнала всех телефонных звонков, поступивших с центрального телефонного коммутатора. Или, возможно, вы установили утилиту mod_log_sql для Apache и теперь можете хранить сведения обо всех посещениях сайта прямо в таблице. В таких приложениях, вероятно, самым важным является обеспечение быстродействия.

Вы же не хотите, чтобы база данных оказалась узким местом. Подсистемы хранения данных MyISAM и Archive будут очень хорошо работать, поскольку характеризуются небольшими издержками, и вы сможете осуществлять тысячи операций записи в секунду.

Однако все становится гораздо интереснее, когда приходит время генерировать отчеты на основе записанных в журнал данных. В зависимости от того, какие запросы вы используете, велика вероятность, что сбор данных для отчета значительно замедлит процесс добавления новых записей. Что можно сделать в этой ситуации?

Например, можно использовать встроенную функцию репликации MySQL для дублирования данных на второй сервер, где затем будут запущены запросы, активно потребляющие ресурсы центрального процессора (ЦП). Таким образом, главный сервер останется свободным для вставки записей и вы сможете делать любые запросы, не беспокоясь о том, как создание отчета повлияет на ведение журнала в реальном времени.

Также вы можете запускать запросы в периоды низкой загрузки, правда, по мере развития приложения эта стратегия может стать неработоспособной.

Другой вариант — вести журнал в таблице, имя которой составлено из года и названия или номера месяца, например `web_logs_2008_01` или `web_logs_2008_jan`. Если вы будете адресовать запросы к таблицам, в которые уже не производится запись, то приложение сможет непрерывно сохранять новые данные журнала в текущую таблицу.

Таблицы только для чтения или преимущественно для чтения

Таблицы с данными, которые используются для создания каталога или списка (вакансии, аукционы, недвижимость и т. п.), обычно характеризуются тем, что считывание из них происходит значительно чаще, чем запись в них. С такими таблицами хорошо применять MyISAM — *если не думать о том, что происходит при ее сбое*. Но не стоит недооценивать важность этого фактора. Многие пользователи не понимают, как рискованно применять подсистему хранения, которая даже не пытается извлечь данные, записанные на диск. (MyISAM просто записывает данные в память и предполагает, что операционная система сбросит их на диск позже.)

Очень полезно запустить имитацию реальной нагрузки на тестовом сервере, а затем в прямом смысле слова выдернуть вилку из розетки. Личный опыт восстановления данных после сбоя бесценен. Он убережет от неприятных сюрпризов в будущем.

Не стоит слепо доверять народной мудрости сообщества, которая гласит: «MyISAM быстрее, чем InnoDB». Категоричность этого утверждения спорна. Мы можем перечислить десятки ситуаций, когда InnoDB на голову опережает MyISAM, особенно

в приложениях, где применяются кластерные индексы или данные целиком размещаются в памяти. По мере дальнейшего чтения вы начнете понимать, какие факторы влияют на производительность подсистемы хранения (размер данных, требуемое количество операций ввода/вывода, первичные ключи и вторичные индексы и т. п.) и какие из них более значимы в вашем приложении.

Проектируя подобные системы, мы используем InnoDB. Сначала MyISAM может произвести впечатление хорошо работающей подсистемы, но при большой нагрузке она просто рухнет. Все будет заблокировано, и при сбое сервера вы потеряете данные.

Обработка заказов

При обработке заказов практически всегда требуются транзакции. Созданный наполовину заказ вряд ли обрадует ваших клиентов. Важно также определить, поддерживает ли подсистема ограничения внешнего ключа. Для приложений обработки заказов оптимальным выбором является InnoDB.

Доски объявлений и дискуссионные форумы

Тематические дискуссии являются интересной задачей для пользователей MySQL. Существуют сотни бесплатных систем на основе языков PHP и Perl, которые позволяют организовывать тематические дискуссии. Многие из них не умеют эффективно использовать базу данных, в результате чего для каждого обслуживаемого обращения в них запускается множество запросов. Некоторые из них разработаны, чтобы обеспечить независимость от используемой базы данных, поэтому их запросы не извлекают должной выгоды из возможностей конкретной СУБД. Подобные системы также зачастую обновляют счетчики и собирают статистику по различным дискуссиям. Большинство из них использует для хранения всего объема данных лишь несколько монолитных таблиц. В результате несколько основных таблиц оказываются перегруженными операциями записи и чтения, и возникает значительная конкуренция блокировок, необходимых для обеспечения целостности данных.

Несмотря на недостатки проектирования, большинство таких систем хорошо работают при малых и средних нагрузках. Однако если сайт становится довольно большим и генерирует значительный трафик, скорость его работы существенно снижается. Очевидным решением этой проблемы является переход на другую подсистему хранения данных, которая может обслуживать больше операций чтения и записи. Но иногда пользователи, поступающие подобным образом, обнаруживают, что система начинает работать еще медленнее!

Эти пользователи не учитывают, что приложение запускает довольно специфические запросы, например, вот такого вида:

```
mysql> SELECT COUNT(*) FROM table;
```

Проблема заключается в том, что не все подсистемы хранения могут быстро выполнить подобные запросы: MyISAM на это способна, а другие — не всегда.

Похожие примеры можно привести для каждой подсистемы. Следующие главы помогут вам избежать неприятных сюрпризов и разобраться в том, как выявлять и решать проблемы такого рода.

Приложения на CD

Если вам когда-нибудь потребуется распространять приложения, использующие файлы данных MySQL, на CD или DVD, подумайте о применении таблиц типа MyISAM или сжатых таблиц MyISAM, которые можно легко изолировать и скопировать на другой носитель. Сжатые таблицы MyISAM занимают значительно меньше места, чем несжатые, но они предназначены только для чтения. В некоторых приложениях это может стать проблемой, но, поскольку данные все равно предназначены для записи на носитель, поддерживающий только чтение, нет оснований избегать использования сжатых таблиц для этой конкретной задачи.

Большие объемы данных

Слишком много — это сколько? Мы проектировали (и управляли) — или помогали проектировать (и управлять) — множество баз данных размером от 3 до 5 Тбайт или даже больше, работавших с подсистемой InnoDB. И это на одном сервере, без сегментирования. Это вполне осуществимо, но нужно с умом подойти к выбору оборудования, а также запланировать для сервера возможность справляться с большим объемом операций ввода/вывода. При таких размерах крах MyISAM становится настоящей катастрофой.

Если вы предусматриваете значительный размер базы данных, например десятки терабайт, то, вероятно, проектируете хранилище данных. В этом случае подсистема хранения Infobright будет лучшим выбором. Очень большие базы данных, для которых не подходит Infobright, вероятно, могут использовать TokuDB.

Преобразования таблиц

Существует несколько способов преобразования таблицы из одной подсистемы хранения в другую, и у каждого из них есть свои преимущества и недостатки.

В следующих разделах мы рассмотрим три наиболее распространенных способа.

ALTER TABLE

Простейший способ преобразования таблицы из одной подсистемы в другую — использование команды `ALTER TABLE`. Следующая команда преобразует таблицу `mytable` к типу InnoDB:

```
mysql> ALTER TABLE mytable ENGINE = InnoDB;
```

Такой синтаксис работает для всех подсистем хранения, но есть одна загвоздка — это может занять много времени. MySQL будет построчно копировать старую таблицу в новую. В это время, скорее всего, будет задействована вся пропускная способность диска сервера, а исходная таблица окажется заблокированной для чтения. Поэтому будьте осторожны, применяя этот подход для активно используемых таблиц. Вместо него можно задействовать один из рассмотренных далее методов, которые сначала создают копию таблицы.

При изменении подсистемы хранения утрачиваются все присущие старой подсистеме возможности. Например, после преобразования таблицы InnoDB в MyISAM, а потом обратно будут потеряны все внешние ключи, определенные в исходной таблице InnoDB.

Экспорт и импорт

Чтобы лучше контролировать процесс преобразования, можете сначала экспортировать таблицу в текстовый файл с помощью утилиты mysqldump. После этого можно будет просто изменить команду `CREATE TABLE` в этом текстовом файле. Не забудьте изменить название таблицы и ее тип, поскольку нельзя иметь две таблицы с одинаковыми именами в одной и той же базе данных, даже если у них разные типы, — а mysqldump по умолчанию пишет команду `DROP TABLE` перед командой `CREATE TABLE`, так что вы можете потерять свои данные, если не будете осторожны!

CREATE и SELECT

Третий способ преобразования обеспечивает компромисс между скоростью первого и безопасностью второго. Вместо того чтобы экспортировать всю таблицу или преобразовывать ее за один прием, создайте новую таблицу и используйте для ее заполнения команду MySQL `INSERT ... SELECT` следующим образом:

```
mysql> CREATE TABLE innodb_table LIKE myisam_table;
mysql> ALTER TABLE innodb_table ENGINE=InnoDB;
mysql> INSERT INTO innodb_table SELECT * FROM myisam_table;
```

При небольших объемах данных это хороший вариант. Но если объем данных велик, то зачастую гораздо эффективнее заполнять таблицу частями, подтверждая транзакцию после каждой части, чтобы журнал отмены не становился слишком большим. Предполагая, что `id` является первичным ключом, запустите этот запрос несколько раз (каждый раз задавая все бóльшие значения `x` и `y`), пока не скопируете все данные в новую таблицу:

```
mysql> START TRANSACTION;
mysql> INSERT INTO innodb_table SELECT * FROM myisam_table
    -> WHERE id BETWEEN x AND y;
mysql> COMMIT;
```

После выполнения у вас будет исходная таблица, которую можно удалить, и целиком заполненная новая таблица. Не забудьте заблокировать исходную таблицу, если не хотите получить несогласованную копию данных!

Хронология MySQL

История версий MySQL поможет разобраться в том, с какой именно версией сервера вам лучше работать. А если вы уже давно пользуетесь ею, наверняка будет интересно вспомнить, как развивалась эта система.

- *Версия 3.23 (2001).* Этот релиз MySQL считается первым по-настоящему жизнеспособным вариантом для широкого применения. В этом варианте MySQL еще не сильно превосходила язык запросов на основе неструктурированных файлов, но MyISAM была представлена в качестве замены ISAM — более старой и гораздо более ограниченной подсистемы хранения. Также стала доступна подсистема InnoDB, но из-за своей новизны она не была включена в стандартный двоичный дистрибутив. Чтобы использовать InnoDB, нужно было скомпилировать сервер самостоятельно. В версии 3.23 были добавлены полнотекстовая индексация и репликация. Последняя была призвана стать той самой отличительной особенностью, которая сделала бы MySQL известной в качестве базы данных, по большей части управляемой через Интернет.
- *Версия 4.0 (2003).* Появилась новая синтаксическая функция поддержки работы команд `UNION` и `DELETE` с несколькими таблицами. Репликация была переписана для использования двух потоков на реплике вместо одного потока, который выполнял всю работу и нес убытки от переключения задачи. InnoDB была запущена как стандартная составляющая сервера с полным набором отличительных характеристик, таких как строковая блокировка, внешние ключи и т. д. Кэширование запросов было введено в версии 4.0 (и с тех пор не изменилось). Также была представлена поддержка SSL-соединений.
- *Версия 4.1 (2005).* Были добавлены дополнительные функции синтаксиса запросов, включая подзапросы и команду `INSERT ON DUPLICATE KEY UPDATE`. Начала поддерживаться кодировка UTF-8. Появились новый двоичный протокол и поддержка предварительно подготовленных операторов.
- *Версия 5.0 (2006).* В этой версии появилось несколько «корпоративных» функций: представления, триггеры, хранимые процедуры и хранимые функции. Подсистема хранения данных ISAM полностью удалена, введены новые подсистемы хранения, например Federated.
- *Версия 5.1 (2008).* Этот релиз стал первым после того, как Sun Microsystems поглотила MySQL AB; работа над ним продолжалась более пяти лет. В версии 5.1 были представлены сегментирование, построчная репликация и разнообразные API для плагинов, включая API подключаемой подсистемы хранения. Подсистема хранения BerkeleyDB, первая транзакционная подсистема хранения в MySQL, удалена, а некоторые другие, такие как Federated, устарели. Кроме того, компания Oracle, которая к этому моменту уже владела Innobase Oy[1], выпустила подключаемую подсистему хранения InnoDB.

[1] Сейчас Oracle владеет также BerkeleyDB.

- *Версия 5.5 (2010).* MySQL 5.5 была первой версией, выпущенной после того как Oracle поглотила Sun (и, следовательно, MySQL). Основное внимание было уделено улучшению производительности, масштабируемости, репликации, сегментированию и поддержке Microsoft Windows, внесено и множество других улучшений. InnoDB стала подсистемой хранения, установленной по умолчанию, а многие устаревшие функции, настройки и параметры были удалены. Добавлены база данных `PERFORMANCE_SCHEMA`, а также первая партия расширенного инструментария. Добавлены новые API плагинов для репликации, аутентификации и аудита. Появилась поддержка полусинхронного механизма репликации, и в 2011 году Oracle выпустила коммерческие плагины для аутентификации и организации пула потоков. Внесены значительные изменения в архитектуру InnoDB, например, появился разделенный буферный пул.
- *Версия 5.6 (2013).* MySQL 5.6 приобрела множество новых функций, в том числе впервые за многие годы появились значительные улучшения оптимизатора запросов, больше плагинов API (например, один для полнотекстового поиска), улучшения механизма репликации и значительно расширенные инструменты в базе данных `PERFORMANCE_SCHEMA`. В то время как версия MySQL 5.5 в основном стремилась улучшить и исправить базовые функции и содержала немного нововведений, MySQL 5.6 нацелена на серьезное улучшение работы сервера и повышение производительности.
- *Версия 6.0 (отменена).* Версия 6.0 вносит путаницу из-за перекрывающейся нумерации. Она была анонсирована во время разработки версии 5.1. Ходили слухи о большом количестве новых функций, таких как резервные онлайн-копии и внешние ключи на уровне сервера для всех подсистем хранения, улучшение механизма подзапросов и поддержка пула потоков. Этот релиз был отменен, и Sun возобновила разработку версии 5.4, которая в итоге была выпущена как версия 5.5. Многие из возможностей версии 6.0 реализованы в версиях 5.5 и 5.6.

Давайте подведем итоги исторического обзора MySQL: на раннем этапе жизненного цикла она стала прорывной технологией[1]. Несмотря на ограниченные функциональные возможности и второсортные функции, ее отличительные особенности и низкая стоимость сделали ее революционной новинкой, которая взорвала Интернет. В ранних версиях 5.x MySQL попыталась выйти на корпоративный рынок с такими функциями, как представления и хранимые процедуры, но они были нестабильны и содержали ошибки, поэтому не все шло как по маслу. Если вдуматься, то поток исправлений ошибок, допущенных в MySQL 5.0, не иссяк вплоть до релиза 5.0.50, а MySQL 5.1 была ненамного лучше. Выпуск релизов 5.0 и 5.1 был отложен, а поглощения, произведенные Sun и Oracle, заставили многих наблюдателей понервничать. Но, на наш взгляд, развитие идет полным ходом: версия MySQL 5.5 стала релизом

[1] Термин «прорывные технологии» появился в книге Клейтона М. Кристенсена «Дилемма инноватора».

самого высокого качества в истории MySQL, Oracle сделала MySQL гораздо более удобной для корпоративных клиентов, а версия 5.6 обещает значительные улучшения функциональности и производительности.

Говоря о производительности, неплохо было бы продемонстрировать базовый эталонный тест изменения производительности сервера с течением времени. Мы решили не тестировать версии старше 4.1, потому что сейчас редко можно встретить 4.0 и более старые. Кроме того, очень сложно сравнивать эталонные тесты разных версий по причинам, о которых вы подробнее прочтете в следующей главе. Мы получили массу удовольствия, создавая метод эталонного тестирования, который будет работать одинаково со всеми тестируемыми версиями сервера, и нам потребовалось предпринять множество попыток, прежде чем мы достигли успеха. В табл. 1.2 показано количество транзакций в секунду для нескольких уровней конкурентного доступа.

Таблица 1.2. Эталонный тест блокировки «только для чтения» нескольких версий MySQL

Потоки	MySQL 4.1	MySQL 5.0	MySQL 5.1	MySQL 5.1 с InnoDB	MySQL 5.5	MySQL 5.6
1	686	640	596	594	531	526
2	1307	1221	1140	1139	1077	1019
4	2275	2168	2032	2043	1938	1831
8	3879	3746	3606	3681	3523	3320
16	4374	4527	4393	6131	5881	5573
32	4591	4864	4698	7762	7549	7139
64	4688	5078	4910	7536	7269	6994

На рис. 1.2 изображен график, который представляет результаты в более наглядном виде.

Прежде чем интерпретировать результаты, следует немного рассказать о самом эталонном тесте. Мы запускали его на машине Cisco UCS C250 с двумя шестиядерными процессорами, каждый из которых имеет два аппаратных потока. Сервер содержит 384 Гбайт ОЗУ, но мы провели тест с объемом данных 2,5 Гбайт, поэтому настроили для MySQL пул буферов размером 4 Гбайт. Эталонным тестом была стандартная рабочая нагрузка «только для чтения» SysBench, причем все данные были в InnoDB, хранились в оперативной памяти и зависели только от быстродействия центрального процессора. Мы провели 60-минутный тест для каждой точки замера, измеряя пропускную способность каждые 10 секунд и используя 900 секунд измерений после прогрева и стабилизации сервера для получения окончательных результатов.

Теперь, глядя на результаты, мы можем заметить две основные тенденции. Во-первых, версии MySQL, которые включают плагин InnoDB, намного лучше работают при более высокой степени конкурентности, то есть они более масштабируемы.

Этого следовало ожидать, так как нам известно, что предыдущие версии серьезно ограничены в плане поддержки конкурентности. Во-вторых, более новые версии MySQL медленнее старых при однопоточных рабочих нагрузках, чего вы, возможно, и не ожидали. Но это легко объяснить, отметив, что нагрузка «только для чтения» — очень простая рабочая нагрузка. Новые версии серверов имеют более сложную грамматику SQL и множество других функций и улучшений, которые позволяют производить более сложные запросы, но также требуют дополнительных издержек для простых запросов, которые мы как раз и использовали в эталонном тесте. Старые версии сервера проще и, следовательно, имеют преимущество при выполнении простых запросов.

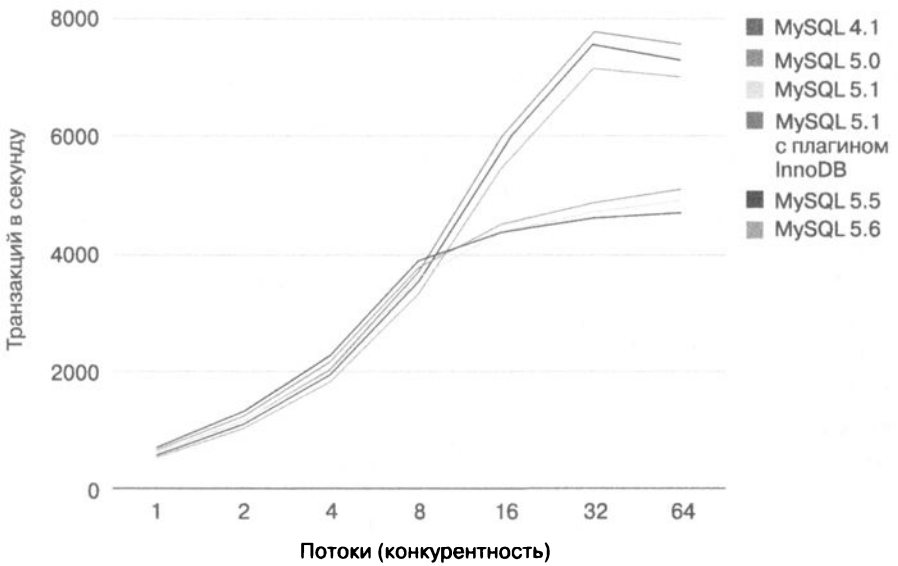


Рис. 1.2. Эталонный тест блокировки «только для чтения» нескольких версий MySQL

Мы хотели показать вам более сложный эталонный тест чтения/записи (например, TPC-C) с более высокой степенью конкурентности, но не смогли сделать это для большого количества столь разных версий сервера. И можем сказать, что в целом новые версии сервера имеют более высокую и более стабильную производительность при сложных нагрузках, особенно при высокой степени конкурентности, и с большим набором данных.

Какую версию лучше использовать вам? Это зависит от особенностей вашего бизнеса больше, чем от технических потребностей. В идеале вы должны ориентироваться на самую новую из доступных версий, но, конечно, можете подождать до тех пор, пока не появится новейшая версия с первыми исправлениями ошибок. Если вы только разрабатываете приложение, то есть еще не выпустили его, можете задуматься о его создании в следующей версии, чтобы максимально отсрочить обновление и увеличить жизненный цикл.

Модель развития MySQL

За многие годы процесс разработки и выпуска MySQL претерпел существенные изменения, но сейчас, похоже, приобрел устойчивый ритм. Oracle периодически выпускает релизы промежуточных этапов разработки с предварительным просмотром функций, которые в конце концов будут включены в следующий GA-релиз[1]. Они предназначены для тестирования и получения обратной связи, а не для использования на производстве, но Oracle заявляет о том, что они высокого качества, и по существу, готовы к выпуску в любое время, и мы не видим причин не соглашаться с этим. Oracle также периодически выпускает лабораторные превью, которые представляют собой специальные сборки, содержащие только избранные функции для оценки заинтересованными сторонами. Нет гарантии, что эти функции будут включены в следующую версию сервера и наконец раз в кои-то веки Oracle соединит вместе функции, которые считает готовыми, и выпустит новый релиз сервера GA.

MySQL остается программным продуктом с открытым исходным кодом и имеет GPL (General Public License, Универсальная общедоступная лицензия), причем полный исходный код (конечно, за исключением коммерчески лицензированных плагинов) доступен лишь сообществу разработчиков. Oracle, похоже, понимает, что было бы неразумно предоставлять разные версии сервера сообществу и бизнес-клиентам. MySQL AB пробовала поступать подобным образом, но получилось, что ее бизнес-клиенты стали подопытными кроликами, так как лишились тестирования и обратной связи от пользователей. Это противоречило интересам корпоративных клиентов, и такая практика была прекращена, когда управление перешло к Sun.

Теперь, когда Oracle выпускает некоторые серверные плагины только для бизнес-клиентов, MySQL целиком и полностью соответствует так называемой модели с открытым ядром. И хотя периодически слышен недовольный шепоток по поводу выпуска коммерческих плагинов для сервера, он исходит от меньшинства и иногда серьезность ситуации оказывается преувеличенной. Большинство пользователей MySQL, которых мы знаем (а мы знаем многих), похоже, не возражают против этого. Коммерчески лицензированные платные плагины приемлемы для тех пользователей, которые действительно в них нуждаются.

В любом случае, коммерческие плагины — это просто плагины. Они не ограничивают функциональность модели разработки, и сервер самодостаточен и без них. Честно говоря, мы ценим то, что Oracle создает большинство функций в виде плагинов. Если бы функции были встроены прямо на сервере без API, выбора бы не было: вы получили бы одну реализацию и были бы ограничены в возможности скомпоновать что-то, что вам подходит лучше. Например, если Oracle все же выпустит функцию полнотекстового поиска InnoDB в качестве плагина, у нее будет возможность использовать тот же API для разработки аналогичного плагина для Sphinx или Lucene, который многие пользователи могут посчитать более полезным. Неплохи и чистые API внутри сервера. Они способствуют повышению качества кода, а кто к этому не стремится?

[1] GA расшифровывается как «общедоступный», что означает «пригодный для производственного применения».

Итоги главы

MySQL имеет многоуровневую архитектуру, на вершине которой находятся серверные службы и функции выполнения запросов, а под ними — подсистемы хранения. Хотя существует множество различных API плагинов, наиболее важным являются API подсистемы хранения данных. Если вы осознаете, что MySQL выполняет запросы, передавая строки вперед и назад по API подсистемы хранения, то вы поняли одну из основ архитектуры сервера.

MySQL создавалась вокруг подсистемы ISAM (позже MyISAM), а многочисленные подсистемы хранения и транзакций были добавлены позже. Это отражают многие специфические особенности сервера. Например, способ, которым MySQL подтверждает транзакции при выполнении команды `ALTER TABLE`, обусловлен архитектурой подсистемы хранения, как и тот факт, что словарь данных хранится в файлах `.frm`. (Кстати, в подсистеме InnoDB нет ничего, что заставило бы команду `ALTER` быть нетранзакционной, — абсолютно все, что делает InnoDB, является транзакционным.)

API подсистемы хранения имеет некоторые недостатки. Иногда возможность выбора не приносит пользы, а появление большого числа подсистем хранения в горячие деньки после выхода MySQL версий 5.0 и 5.1 могло бы предоставить слишком широкий выбор. В конце концов, InnoDB оказалась очень хорошей подсистемой хранения примерно для 95 % или более пользователей (это просто грубое предположение). Другие подсистемы хранения обычно делают все более сложным и ненадежным, хотя есть особые случаи, когда определенно требуется альтернатива.

Поглощение компанией Oracle сначала InnoDB, а затем MySQL привело оба продукта под одну крышу, так что теперь они могут разрабатываться совместно. Кажется, от этого выигрывают все: InnoDB и сам сервер стремительно становятся лучше, MySQL продолжает распространяться под GPL и остается ПО с открытым исходным кодом, сообщество разработчиков и клиенты получают прочную и стабильную базу данных, а сервер — все больше возможностей для расширения и применения.

2 Эталонное тестирование MySQL

Эталонное тестирование (бенчмаркинг) является важным навыком как для новичков MySQL, так и для опытных пользователей. Проще говоря, эталонное тестирование — это создание рабочей нагрузки, предназначенной для того, чтобы подвергнуть систему стрессу. Основная его цель — изучение поведения системы, однако имеются и другие важные причины для выполнения эталонного тестирования, такие как воспроизведение желаемого состояния системы или испытание на отказ нового оборудования. В этой главе мы рассмотрим причины, стратегии, тактику и инструменты эталонного тестирования MySQL и приложений на базе MySQL. Особый упор сделаем на sysbench, потому что это отличный инструмент для эталонного тестирования MySQL.

Зачем нужно эталонное тестирование

Почему эталонное тестирование так важно? Дело в том, что его очень удобно и эффективно использовать для получения ответа на вопрос, что происходит, когда вы заставляете систему работать. Эталонное тестирование может помочь вам понаблюдать за поведением системы под нагрузкой, определить ее пропускную способность, узнать, какие изменения важны, или посмотреть, как приложение работает с разными данными. Эталонное тестирование позволяет создавать вымышленные обстоятельства в дополнение к реальным условиям, с которыми вы можете столкнуться. С его помощью вы можете:

- подтвердить свои предположения о системе и посмотреть, насколько они реалистичны;
- воспроизвести нежелательное поведение системы, которое вы пытаетесь устранить;
- измерить производительность приложения. Если вы не знаете, как быстро оно работает в данный момент, то не можете быть уверены в целесообразности изменений. Также вы можете использовать результаты предыдущих тестов для диагностики неожиданных проблем;

- имитировать нагрузку, превышающую ту, которую испытывает система сейчас, для определения узкого места масштабирования, с которым вы столкнетесь в первую очередь, когда она начнет расти;
- планировать рост системы. Эталонные тесты могут помочь вам оценить, какие оборудование, пропускная способность сети и другие ресурсы вам понадобятся для эмуляции будущей нагрузки. Это поможет уменьшить риски при модернизации или серьезных изменениях приложения;
- проверить, способно ли приложение работать в изменяющихся условиях. Например, вы можете узнать, как оно работает во время единичного пика одновременно работающих пользователей или при другой конфигурации серверов. Или увидеть, как оно обрабатывает другое распределение данных;
- проверить различные конфигурации оборудования, программного обеспечения и операционной системы. Что лучше для вашей системы — RAID 5 или RAID 10? Как изменяется производительность произвольной записи при переключении с дисков ATA на хранилище SAN? Лучше ли масштабируется ядро Linux 2.4, чем ядро Linux 2.6? Увеличит ли производительность обновление версии MySQL? Что будет, если использовать другой механизм хранения данных? Вы можете ответить на эти вопросы с помощью специальных эталонных тестов;
- убедиться, что недавно приобретенное оборудование настроено правильно. Мы не можем сказать, сколько раз использовали эталонные тесты для испытания на отказ новой системы и обнаруживали неправильную конфигурацию или неисправные аппаратные компоненты. Не стоит запускать новый сервер, не проведя на нем эталонного тестирования и полагаясь лишь на слова провайдера или изготовителя оборудования о том, что установлено и как быстро оно должно работать. Тестирование, когда оно возможно, всегда полезно.

Кроме того, вы можете использовать эталонное тестирование для других целей, таких как создание набора ориентированных на конкретное приложение модульных тестов. Однако здесь мы сосредоточимся только на аспектах, связанных с производительностью.

Проблема с эталонным тестированием состоит в том, что оно ненастоящее. Нагрузка, которую вы используете, чтобы подвергнуть систему стрессу, обычно не дотягивает до реальной. Причина этого состоит в следующем: реальные рабочие нагрузки являются недетерминированными, варьирующимися и слишком сложными для понимания. Если вы проводите эталонное тестирование системы с реальной нагрузкой, по его итогам труднее будет сделать точные выводы.

В чем же нагрузка эталонного тестирования нереалистична? Существует множество искусственных показателей для эталонного теста: размер данных, распределение данных и запросов, но самое главное то, что эталонный тест обычно выполняется так быстро, как только возможно, загружая систему настолько сильно, что она показывает себя с худшей стороны. Во многих случаях хотелось бы заставить инструменты эталонного тестирования работать как можно быстрее, но с определенными оговорками, с возможностью останавливаться при необходимости, чтобы поддержать хорошую

производительность. Это было бы особенно полезно для определения максимальной производительности системы. Однако большинство инструментов эталонного тестирования не поддерживает такую возможность. Лучше не забывать, что используемые инструменты налагают ограничения на значимость и полезность полученных результатов.

Непросто использовать эталонные тесты и для планирования вычислительных мощностей. Результаты эталонных тестов зачастую невозможно экстраполировать. Например, предположим, что вы хотите знать, какой рост бизнеса сможете поддерживать с помощью нового сервера базы данных. Вы проводите эталонное тестирование существующего и затем нового сервера и обнаруживаете, что он может выполнять в 40 раз больше транзакций в секунду. Но это не означает, что, используя новый сервер, ваш бизнес тоже сможет увеличиться в 40 раз. К тому времени, когда выручка настолько вырастет, система, вероятно, будет иметь больше трафика, больше пользователей, больше данных и больше взаимосвязей между различными частями данных. Не следует ожидать, что все эти факторы вырастут только в 40 раз, особенно это касается количества взаимосвязей. Кроме того, к этому моменту ваше приложение почти наверняка изменится: появятся новые функции, часть из которых могут влиять на базу данных далеко не пропорционально их кажущейся сложности. Эти изменения рабочей нагрузки, данных, взаимосвязей и функций очень трудно имитировать, а их воздействие сложно спрогнозировать.

В результате мы обычно соглашаемся на приблизительные результаты, чтобы узнать, есть ли у системы достаточное количество резервных мощностей. Можно выполнить более реалистичное тестирование нагрузки, отличающееся от эталонного, но это требует больше внимания при создании набора данных и рабочей нагрузки. И все же это не настоящий эталонный тест. Эталонные тесты проще, лучше сопоставимы друг с другом, дешевле и проще в управлении. И несмотря на их ограничения, эталонные тесты полезны. Вы просто должны четко понимать, что делаете и какой смысл в полученном результате.

Стратегии эталонного тестирования

Существует две основные стратегии эталонного тестирования: можно тестировать приложение целиком или проверить лишь MySQL. Мы назовем эти стратегии соответственно *полностековым* и *покомпонентным* эталонным тестированием. Есть несколько причин для измерения производительности приложения в целом вместо тестирования только MySQL.

- Вы тестируете все приложение, включая веб-сервер, код приложения, сеть и базу данных. Это полезно, поскольку вас интересует производительность не MySQL, а всей системы.
- MySQL не всегда является узким местом приложения, и полное эталонное тестирование позволяет это выявить.
- Только в процессе полного тестирования вы можете увидеть, как ведет себя кэш каждой части.

- Эталонные тесты хороши лишь в той степени, в какой они отражают реальное поведение приложения, чего трудно добиться при тестировании его по частям.

В то же время эталонные тесты приложения сложно создавать и еще сложнее правильно настраивать. Если вы плохо спроектируете эталонный тест, то можете принять ошибочное решение, поскольку полученные в этом случае результаты не отражают реального положения.

Однако иногда нет необходимости получать информацию обо всем приложении. Возможно, нужно просто выполнить эталонное тестирование MySQL, по крайней мере на начальном этапе. Такое эталонное тестирование полезно, если вы хотите:

- сравнить различные схемы или запросы;
- протестировать конкретную проблему, обнаруженную в приложении;
- избежать длительного эталонного тестирования, ограничившись коротким тестом, который позволит быстро внести изменения и измерить их.

Кроме того, эталонное тестирование MySQL полезно, когда вы можете выполнить запросы характерные для своего реального набора данных. Как сами данные, так и размер набора должны быть реалистичными. По возможности используйте мгновенный снимок фактических рабочих данных.

К сожалению, настройка реалистичного эталонного теста может оказаться сложной и трудоемкой задачей, поэтому, если сумеете получить копию рабочего набора данных, считайте себя счастливчиком. Но это может оказаться невозможным — например, вы создаете новое приложение, которым пользуются немногие и в котором еще недостаточно данных. Если вы хотите знать, как оно будет работать, когда увеличится, то у вас нет другого выхода, кроме как сгенерировать больший объем данных и значительную нагрузку.

Что измерять. Перед началом эталонного тестирования нужно определить цели — собственно, это следует сделать даже до начала проектирования тестов. Цели определят инструменты и технику, которые вы будете использовать для получения точных осмысленных результатов. Постарайтесь сформулировать цели в виде вопросов, например: «Лучше ли этот процессор, чем тот?» или «Будут ли новые индексы работать эффективнее, чем нынешние?».

Иногда для измерения разных показателей требуются различные подходы. Так, нельзя определить сетевые задержки и пропускную способность с помощью одних и тех же эталонных тестов.

Рассмотрим следующие показатели и обсудим, как они соответствуют вашим.

- *Пропускная способность.* Определяется как количество транзакций в единицу времени. Это один из классических эталонных тестов приложений баз данных. Стандартизованные эталонные тесты, такие как TPC-C (см. http://www.tpc.org), применяются часто, и многие производители баз данных активно работают над улучшением характеристик своих продуктов, определяемых с их помощью. Эти эталонные тесты измеряют производительность оперативной обработки

транзакций (OLTP) и лучше всего подходят для интерактивных многопользовательских приложений. Общепринятой единицей измерения является количество транзакций в секунду, хотя иногда указывают и транзакции в минуту.

- *Время отклика или задержки.* Этот показатель определяет общее время выполнения задачи. В зависимости от приложения вам может потребоваться измерить время в микро- или миллисекундах, секундах или минутах. Отсюда можно определить среднее, минимальное и максимальное время отклика, а также процентили. Максимальное время отклика редко бывает полезно, поскольку чем дольше эталонный тест работает, тем выше, скорее всего, будет этот показатель. Кроме того, его невозможно повторить, так как значение наверняка будет сильно варьироваться при разных прогонах теста. По этой причине обычно используют *время отклика в процентилях*. Например, если время отклика составляет 5 миллисекунд с 95-м, значит, в 95 % случаев задача будет выполнена за 5 миллисекунд или быстрее.

  Обычно имеет смысл представить результаты эталонных тестов графически: либо в виде линейного графика (например, среднее и 95-й процентиль), либо в виде диаграммы разброса, на которой видно, как распределены результаты. Эти графики показывают, как будут вести себя эталонные тесты при длительных испытаниях. Мы вернемся к этому моменту позднее в данной главе.

- *Параллелизм (конкурентный доступ).* Параллелизм является важным показателем, но его часто неверно толкуют и плохо применяют. Например, число пользователей, просматривающих сайт в конкретный момент, обычно измеряется количеством сеансов[1]. Однако в протоколе HTTP нет сохранения состояния, и если большинство пользователей просто читают содержимое страницы, открытой в браузере, это не вызывает параллелизма на веб-сервере. Аналогично, параллелизм на веб-сервере не обязательно означает параллелизм в базе данных. Единственное, что непосредственно связано с параллелизмом, — это объем данных, с которым должен справляться механизм хранения сеансов. Более точный показатель параллелизма на веб-сервере — это количество одновременных запросов в любой момент времени.

  Кроме того, вы можете измерять параллелизм в различных местах приложения. Более высокий параллелизм на веб-сервере может вызвать более высокий параллелизм на уровне базы данных, однако на него могут повлиять также язык программирования и набор используемых инструментов. Убедитесь, что вы не путаете открытые соединения с сервером базы данных с параллелизмом. Хорошо спроектированное приложение может иметь сотни открытых соединений с сервером MySQL, но только часть из них будут выполнять запросы одновременно. Таким образом, сайт с «50 000 посетителей одновременно» может требовать только 10 или 15 одновременно выполняющихся запросов к серверу MySQL!

  Иными словами, при эталонном тестировании следует озаботиться *рабочим параллелизмом*, то есть количеством потоков или соединений, работающих одновременно. Измерьте, не падает ли производительность, не увеличивается ли

[1] Это, в частности, привело к тому, что многие владельцы сайтов считали, что к ним заходят десятки тысяч пользователей одновременно.

время отклика при возрастании параллелизма. Если это так, то ваше приложение вряд ли сможет выдерживать всплески нагрузки.

Параллелизм — это показатель, кардинально отличающийся от других, таких как пропускная способность и время отклика. Обычно это не результат, а скорее свойство вашего эталонного теста. Вместо измерения уровня параллелизма, которого достигает ваше приложение, вы обычно просите инструмент эталонного тестирования сгенерировать различные уровни параллелизма и определяете производительность приложения в таких условиях. Однако вы также должны измерить параллелизм в базе данных. Когда запустите sysbench с 32, 64 и 128 потоками, проверьте сервер базы данных во время каждого прогона и запишите значение переменной состояния `Threads_running`. В главе 11 рассмотрим, почему это полезно для планирования производительности.

- *Масштабируемость.* Измерение масштабируемости полезно для систем, в которых необходимо поддерживать уровень производительности в условиях меняющейся нагрузки.

  Подробно мы обсудим масштабируемость в главе 11, но сейчас дадим короткое определение. Идеальная система должна выполнить в два раза больше работы (соответственно в два раза увеличится пропускная способность), если вы удвоите число работников, пытающихся выполнить задачу. Вторая точка зрения на ту же проблему состоит в том, что если вы удвоите доступные ресурсы (например, используете вдвое больше процессоров), то сможете добиться удвоенной пропускной способности. В обоих случаях также требуется, чтобы производительность (время отклика) была приемлемой. Но большинство систем не являются линейно масштабируемыми и демонстрируют уменьшающуюся отдачу и ухудшение производительности при изменении параметров.

  Измерение масштабируемости полезно для планирования пропускной способности, поскольку оно может показать слабые места в вашем приложении, которые не будут видны при других стратегиях эталонного тестирования. Например, если вы спроектировали свою систему для наилучшего выполнения эталонного теста времени отклика с одним соединением (это не лучшая стратегия эталонного тестирования), приложение может плохо работать при другом уровне параллелизма. Эталонный тест, который измеряет последовательное время отклика при увеличивающемся числе подключений, выявит этот недостаток проектирования.

  Некоторые действия, такие как работа в пакетном режиме для создания сводных таблиц из детализированных данных, требуют короткого времени отклика. Необходимо выполнить их эталонное тестирование на время отклика, но при этом не забыть продумать, как они будут взаимодействовать с другими сеансами. Пакетные задания могут негативно повлиять на интерактивные запросы, и наоборот.

В конечном счете, лучше всего провести эталонное тестирование того, что важно для пользователей. Попытайтесь определить некоторые требования (формально или неформально): каково приемлемое время отклика, какой уровень параллелизма ожидается и т. д. Затем попробуйте спроектировать свои тесты для удовлетворения всех требований, избегая туннельного видения, то есть не сосредотачиваясь на одних моментах и исключая других.

Тактики эталонного тестирования

Закончив с общими вопросами, давайте перейдем к конкретным — к тому, как проектировать и выполнять эталонные тесты. Однако прежде, чем обсуждать, как правильно выполнять эталонное тестирование, рассмотрим распространенные ошибки, которые могут привести к неподходящим или неточным результатам.

- Использование меньшего объема данных, чем требуется, например 1 Гбайт данных, когда приложение должно будет обрабатывать сотни гигабайт. Или использование текущего набора данных, если предполагается, что приложение будет быстро расти.
- Применение данных с неправильным распределением, например равномерно распределенных, если в реальных данных будут встречаться горячие точки. (Сгенерированные случайным образом данные почти всегда имеют нереалистичное распределение.)
- Использование нереалистично распределенных параметров, скажем, в предположении, что профили всех пользователей будут просматриваться с одинаковой частотой[1].
- Применение однопользовательского сценария для многопользовательского приложения.
- Эталонное тестирование распределенного приложения на единственном сервере.
- Несоответствие реальному поведению пользователя, например то, что не учитывается время обдумывания на веб-странице. Реальные пользователи запрашивают страницу, а потом читают ее, они не щелкают на ссылках без остановки.
- Выполнение идентичных запросов в цикле. Реальные запросы не идентичны, так что они могут вызывать ошибки кэша. Идентичные запросы будут полностью или частично кэшированы на каком-то уровне.
- Отсутствие контроля ошибок. Если результаты эталонного теста не имеют смысла — например, медленная операция внезапно очень быстро заканчивается, — ищите ошибку. Возможно, вы просто протестировали, насколько быстро MySQL может обнаружить синтаксическую ошибку в запросе SQL! Возьмите за правило после выполнения теста проверять журнал ошибок.
- Игнорирование проверки работы системы, когда она не прогрета, например сразу после перезагрузки. Иногда нужно знать, как быстро ваш сервер наберет полную мощность после перезагрузки, таким образом, потребуется специально проверить время разогрева. И наоборот, если вы предполагаете изучить производительность в нормальном режиме, имейте в виду, что при проведении эталонного тестирования сразу после рестарта кэш будет не разогрет и результаты будут отличаться от полученных при разогретом кэше.
- Использование установок сервера по умолчанию. Оптимизация настроек сервера описана в следующих главах.
- Тестирование выполняется слишком быстро. Учтите, что тест должен длиться некоторое время. Мы еще поговорим об этом позже.

[1] Джастин Бибер, мы тебя любим! Шутка.

Простое предотвращение этих ошибок поможет вам значительно улучшить качество результатов.

При прочих равных условиях необходимо стараться делать тесты настолько реалистичными, насколько это возможно. Однако иногда имеет смысл применять немного нереалистичные эталонные тесты. Предположим, что приложение находится не на том же компьютере, где размещен сервер базы данных. Реалистичнее запустить эталонные тесты в той же конфигурации, но это добавит факторы, показывающие, например, насколько быстра сеть и какова ее загрузка. Эталонное тестирование производительности на одном узле обычно проще и в некоторых случаях дает достаточно точные результаты. Вы должны сами решать, когда такой подход допустим.

Проектирование и планирование эталонных тестов

Первым шагом при планировании эталонных тестов является определение проблемы и цели. Затем следует решить, использовать ли стандартный эталонный тест или разработать собственный.

Если вы используете стандартный эталонный тест, выберите тот, который соответствует вашим потребностям. Например, не применяйте эталонный тест TPC-H для системы электронной коммерции. По словам создателей, «TPC-H — это специальное решение, поддерживающее эталонное тестирование». Следовательно, такие тесты не подходят для OLTP-систем.

Разработка собственного эталонного теста является сложным цикличным процессом. Для начала сделайте мгновенный снимок рабочего набора данных. Убедитесь, что можете восстановить эти данные для последующих запусков теста.

Затем вам потребуются запросы к данным. Вы можете превратить комплект модульных тестов в простейший эталонный тест, просто прогнав его много раз, но вряд ли это соответствует тому, как реально используется база данных. Лучше записать все запросы в рабочей системе в течение репрезентативного отрезка времени, например в течение часа в период пиковой нагрузки или в течение всего дня. Если будут записаны запросы за малый промежуток времени, возможно, вам потребуется несколько таких периодов. Это охватит все действия системы, такие как запросы для еженедельного отчета или работа в пакетном режиме, которую вы запланировали на часы минимальной загрузки[1].

Вы можете записывать запросы на нескольких уровнях. Например, если требуется выполнить эталонное полностековое тестирование, то можно протоколировать HTTP-запросы на веб-сервере. Или включить журнал запросов MySQL. Но если вы будете воспроизводить запросы из журнала, не забудьте создать отдельные потоки, а не просто последовательно повторяйте запросы один за другим. Также важно создавать отдельный поток в журнале для каждого соединения вместо хаотичных

[1] Конечно, все это важно, если вы хотите выполнить идеальное эталонное тестирование. Но реальная жизнь обычно вносит коррективы.

запросов в разных потоках. В журнале запросов видно, на каком соединении был выполнен каждый запрос.

Даже если вы не создаете собственный эталонный тест, все равно следует составить план тестирования. Вам придется выполнять эталонные тесты много раз, поэтому необходимо иметь возможность точно их повторять. Составьте также план на будущее: возможно, в следующий раз тестировать будет кто-то другой. Но даже если это будете вы, то, возможно, не сможете точно вспомнить, как именно делали это в первый раз. План должен включать в себя тестовые данные, шаги, предпринятые для настройки системы, информацию об измерениях и анализе результатов и план прогрева сервера.

Разработайте методику документирования параметров и результатов и тщательно протоколируйте каждый прогон. Ваш метод документирования может быть и очень простым, как, например, запись в электронную таблицу или блокнот, и сложным, как создание специально спроектированной базы данных. Вероятно, вы захотите написать какие-то скрипты для упрощения анализа результатов, поскольку чем проще будет их обрабатывать, не открывая электронную таблицу и текстовые файлы, тем лучше.

Должно ли быть длительным эталонное тестирование?

Важно, чтобы эталонное тестирование занимало разумное время. Если вы заинтересованы в стабильной работе системы, а наверняка это так, то необходимо наблюдать ее в устойчивом состоянии. Это может потребовать неожиданно много времени, особенно на серверах с большим количеством данных и объемной памятью. В большинстве систем есть несколько буферов, которые создают наращиваемую мощность — способность амортизировать пики, откладывая некоторую работу и выполняя ее позже, после того как нагрузка схлынет. Но если вы задействуете эти механизмы в течение длительного времени, буферы переполнятся и вы в конце концов увидите, что система не может выдерживать краткосрочную максимальную нагрузку.

Иногда неизвестно, как долго должен выполняться эталонный тест. В этом случае можете просто запустить его, не ограничивая время, и наблюдать, пока не удостоверитесь, что система начала стабилизироваться. Далее приведен пример того, как мы это делали в незнакомой системе. На рис. 2.1 показан график временных рядов пропускной способности чтения и записи на диск.

По мере прогрева системы (после 3–4 часов работы) линия, характеризующая процесс чтения, стала устойчивой, тогда как линия, показывающая запись, демонстрировала резкие колебания на протяжении по меньшей мере 8 часов. И даже после этого на графике есть несколько резких колебаний. В дальнейшем процессы как чтения, так и записи, судя по всему, стабилизировались[1]. Эмпирическое правило гласит: ждите,

[1] Кстати, график, характеризующий запись, демонстрирует очень плохие результаты, устойчивое состояние этой системы — катастрофическая производительность. Называть это устойчивым состоянием почти смешно, тем не менее мы полагаем, что получили характеристику поведения сервера в долгосрочной перспективе.

пока система не станет выглядеть устойчивой, по крайней мере на протяжении времени, требующегося для ее разогрева. Мы проводили этот эталонный тест в течение 72 часов, чтобы гарантировать, что получили характеристику поведения системы в долгосрочной перспективе.

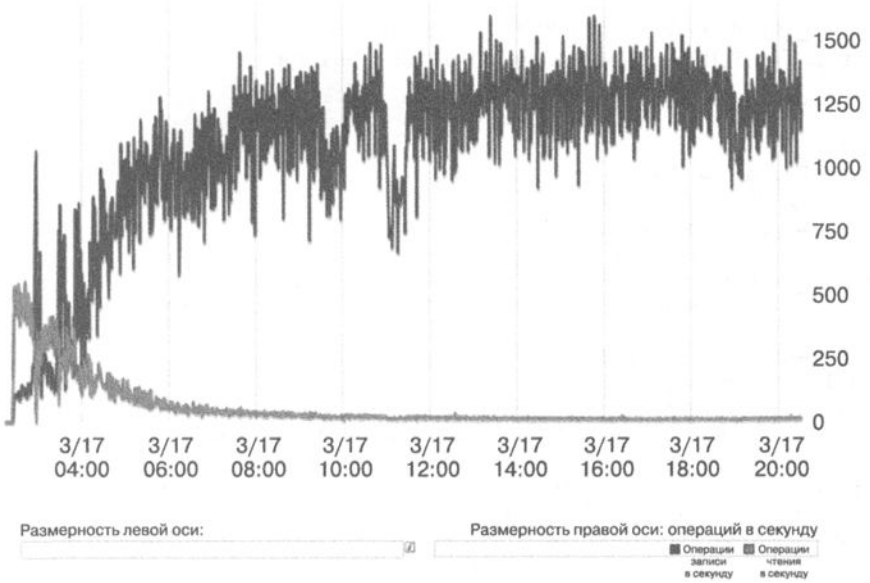


Рис. 2.1. Производительность ввода/вывода во время расширенного эталонного тестирования

Очень распространенная ошибка при эталонном тестировании заключается в том, чтобы запустить серию коротких тестов, например по 60 секунд, и на их основании сделать вывод о производительности системы. Мы слышим много комментариев, таких как «Я попытался провести эталонное тестирование новой версии сервера, и она оказалась не быстрее старой». Изучая реальные эталонные тесты, мы часто обнаруживаем, что они выполнены таким способом, на основании которого нельзя делать подобные выводы. Иногда люди говорят, что у них просто нет времени для эталонного тестирования в течение 8 или 12 часов на десяти различных уровнях параллелизма на двух или трех версиях сервера. Если у вас нет времени для правильного эталонного тестирования, то время на него вы потратили впустую: лучше доверять чужому опыту, чем делать неполный эталонный тест и получать неправильные результаты.

Фиксация производительности и состояния системы

Важно собрать во время эталонного тестирования как можно больше информации о тестируемой системе (SUT). Грамотный подход состоит в создании базового каталога с подкаталогами для результатов каждого запуска. После этого можете поместить результаты, файлы конфигурации, измерения, скрипты и заметки для каждого запуска в соответствующий подкаталог. Если вы получили больше данных, чем вас

интересует, запишите их. Гораздо лучше иметь ненужные данные, чем пропустить что-то важное: возможно, дополнительные данные пригодятся в будущем. Попробуйте записать показатели состояния и производительности, такие как использование ЦП, дисковый ввод/вывод, статистика сетевого трафика, счетчики из `SHOW GLOBAL STATUS` и т. д.

Приведем пример сценария оболочки, который вы можете использовать для сбора данных по MySQL во время эталонного тестирования:

```
#!/bin/sh

INTERVAL=5
PREFIX=$INTERVAL-sec-status
RUNFILE=/home/benchmarks/running
mysql -e 'SHOW GLOBAL VARIABLES' >> mysql-variables
while test -e $RUNFILE; do
    file=$(date +%F_%I)
    sleep=$(date +%s.%N | awk "{print $INTERVAL - (\$1 % $INTERVAL)}")
    sleep $sleep
    ts="$(date +"TS %s.%N %F %T")"
    loadavg="$(uptime)"
    echo "$ts $loadavg" >> $PREFIX-${file}-status
    mysql -e 'SHOW GLOBAL STATUS' >> $PREFIX-${file}-status &
    echo "$ts $loadavg" >> $PREFIX-${file}-innodbstatus
    mysql -e 'SHOW ENGINE INNODB STATUS\G' >> $PREFIX-${file}-innodbstatus &
    echo "$ts $loadavg" >> $PREFIX-${file}-processlist
    mysql -e 'SHOW FULL PROCESSLIST\G' >> $PREFIX-${file}-processlist &
    echo $ts
 done
 echo Exiting because $RUNFILE does not exist.
```

Этот простой сценарий оболочки является хорошим фреймворком для сбора данных о производительности и состоянии. Есть несколько моментов, которые мы считаем полезными, но которые, возможно, вы не оцените, пока не запустите большие эталонные тесты на многих серверах и не поймете, как трудно отвечать на вопросы о поведении системы.

- Итерации синхронизированы так, что будут запускаться каждый раз, когда время будет без остатка делиться на 5 секунд. Если вы просто напишете в цикле `sleep 5`, циклу потребуется для запуска немногим более 5 секунд и вы не сможете легко сопоставлять данные, полученные этим скриптом, с данными других скриптов или графиков. И даже если ваши циклы каким-то образом будут длиться ровно 5 секунд, то получение данных из одной системы с отметкой времени 15:32:18.218192, а из другой системы — 15:32:23.819437 не очень желательно. Вы, если хотите, можете изменить 5 секунд на другой интервал, например 1, 10, 30 или 60 секунд (мы обычно указываем 5 или 10 секунд).
- Для названия файлов используются дата и время выполнения теста. Когда эталонное тестирование продолжается в течение нескольких дней и файлы становятся большими, вам может потребоваться удалить предыдущие файлы с сервера и освободить место на диске, а также приступить к анализу полных результатов. Когда вы ищете данные, полученные в конкретный момент времени, намного

удобнее использовать для поиска название файла, в котором указано это время, чем искать их в одном файле, размер которого вырос до нескольких гигабайт.

- Каждая выборка начинается с отдельной строки, содержащей метку времени, поэтому вы можете искать выборки в файлах, ориентируясь на конкретный временной интервал. Также можно написать маленькие скрипты awk и sed.
- Скрипт не обрабатывает и не фильтрует все то, что он собирает. Это правильный подход: собрать все в необработанной форме, а затем обработать и отфильтровать. Если будет выполнена предварительная обработка, то позже, когда вы обнаружите аномалию, вам наверняка потребуется больше данных, причем необработанных, чтобы ее понять.
- Когда эталонный тест будет пройден, вы можете завершить выполнение скрипта, удалив в нем файл `/home/benchmarks/running`.

Это всего лишь короткий фрагмент кода, который в текущем виде вряд ли удовлетворит вашим запросам, однако он иллюстрирует правильный подход к сбору данных о производительности и состоянии. Скрипт фиксирует лишь несколько видов данных в MySQL, но вы легко можете добавить к нему больше элементов. Например, можете фиксировать `/proc/diskstats` для записи дискового ввода/вывода с целью последующего анализа с помощью инструмента pt-diskstats[1].

Получение точных результатов

Наилучший способ получить точные результаты — спроектировать эталонный тест таким образом, чтобы он отвечал именно на те вопросы, которые вы хотите задать. Тот ли эталонный тест вы выбрали? Собирает ли он данные, которые нужны для ответа на поставленный вопрос? Не проводится ли эталонное тестирование по неправильным критериям? Например, не запускаете ли вы тест, нагружающий процессор, для прогнозирования производительности приложения, которое, как вы знаете, будет ограничено пропускной способностью систем ввода/вывода?

Затем убедитесь в повторяемости результатов эталонного тестирования. Попытайтесь удостовериться, что система находится в одном и том же состоянии перед каждым прогоном теста. Если эталонный тест особо важен, каждый раз перед его запуском перезагружайте компьютер. Если нужно провести эталонное тестирование на прогретом сервере, что является нормой, то необходимо убедиться, что сервер прогревался достаточно долго (см. предыдущий раздел о том, как долго следует выполнять эталонное тестирование) и что эта процедура воспроизводима. Так, если прогрев состоит из случайных запросов, то результаты теста нельзя будет повторить.

Если тест изменяет данные или схему, восстанавливайте их из снимка между прогонами теста. Вставка в таблицу с тысячей строк не даст того же результата, что вставка в таблицу с миллионом строк! Фрагментация и определенное расположение данных

[1] Подробнее ознакомиться с инструментом pt-diskstats можно в главе 9.

на диске также могут сделать результаты невоспроизводимыми. Один из способов убедиться, что физическое расположение будет почти одинаковым, — быстро отформатировать и скопировать сегмент.

Следите за внешней нагрузкой, системами профилирования и мониторинга, подробной записью в журналы, периодическими заданиями и прочими факторами, которые могут исказить результаты. Типичными сюрпризами являются запуск задания cron в середине теста, или периодическое автоматическое «патрулирование», или запланированная проверка целостности RAID-массива. Убедитесь, что все необходимые для эталонного теста ресурсы на время его исполнения выделены только ему. Если сеть загружена еще какой-то работой или эталонный тест запущен на SAN-устройстве, которое одновременно используется и другими серверами, результаты могут оказаться неточными.

Постарайтесь менять как можно меньше параметров при каждом выполнении эталонного тестирования. Если вы измените сразу несколько факторов, то рискуете что-то упустить. Кроме того, параметры могут зависеть друг от друга, поэтому иногда их нельзя изменять независимо. В ряде случаев вы можете даже не подозревать о существовании такой зависимости, что лишь усложняет дело[1].

Лучше всего менять параметры тестирования итеративно, а не вносить значительные изменения между прогонами. Например, если вы пытаетесь настроить сервер на специфическое поведение, используйте технику «разделяй и властвуй» (уменьшайте или увеличивайте значение параметра вдвое на каждой последующей итерации тестирования).

Мы видели множество эталонных тестов, которые пытаются предсказать производительность после миграции, например, с Oracle на MySQL. Зачастую результаты этих тестов ненадежны, поскольку MySQL эффективно работает с совершенно иными типами запросов, чем Oracle. Если вы захотите узнать, насколько хорошо написанное для Oracle приложение будет работать после миграции на MySQL, то вам, скорее всего, придется перепроектировать схему и запросы с учетом специфики MySQL. (Возможно, иногда, например при разработке кросс-платформенного приложения, вам понадобится узнать, как одни и те же запросы будут работать на обеих платформах, но такое случается редко.)

В любом случае нельзя получить осмысленные результаты с параметрами настройки MySQL по умолчанию, поскольку они рассчитаны на небольшие приложения, которые потребляют очень мало памяти. Самый большой стыд мы испытывали, когда кто-то публиковал ошибочные эталонные тесты, сравнивающие MySQL с другими системами управления реляционными базами данных (СУРБД), с настройками по умолчанию. Кажется, что эталонные тесты новичков часто становятся главными новостями.

---

[1] Это не всегда имеет значение. Например, если вы думаете о переходе с системы Solaris, работающей с оборудованием SPARC, на платформу x86 под управлением GNU/Linux, то нет никакого смысла проводить эталонное тестирование системы на платформе Solaris/x86 в качестве промежуточного шага!

Твердотельные хранилища данных (SSD и PCIe-карты) создают особые проблемы для эталонного тестирования, о чем мы поговорим в главе 9.

Наконец, если вы получите странный результат, не следует просто отклонять его или говорить, что вы не понимаете его. Разберитесь и постарайтесь выяснить, что произошло. Возможно, вы обнаружите ценную информацию, серьезную проблему или дефект, возникший при проектировании эталонного теста. Не стоит публиковать эталонные тесты, если вы не понимаете их результатов. Мы видели немало случаев, когда эталонные тесты со странными результатами оказывались совершенно бессмысленными из-за досадной ошибки и в итоге тестировщик выглядел довольно глупо[1].

Прогон эталонного теста и анализ результатов

После того как вы все подготовили, пора запускать тест, чтобы начать сбор и анализ данных.

Обычно имеет смысл автоматизировать прогоны эталонного теста. Это улучшит результаты и их точность, поскольку не позволит забыть о каких-то шагах и исключит различия в методике прогона теста. А заодно поможет документировать весь процесс.

Подойдет любой метод автоматизации, например Makefile или набор пользовательских скриптов. Выберите любой устраивающий вас язык для написания скриптов: shell, PHP, Perl и т. д. Попытайтесь наиболее полно автоматизировать процедуру тестирования, включая загрузку данных, прогрев системы, запуск эталонного теста и запись результатов.

Правильно настроенное эталонное тестирование может стать одношаговым процессом. Если вы просто запускаете одноразовый эталонный тест, чтобы что-то быстро проверить, автоматизировать его нет смысла. Однако если предполагаете вернуться к его результатам в будущем, обязательно автоматизируйте его. Не сделав этого, вы никогда не вспомните, как запускали эталонный тест и какие параметры указывали, так что в дальнейшем не сможете воспользоваться результатами эталонного тестирования.

Обычно эталонный тест выполняют несколько раз. Конкретное количество итераций зависит от методологии оценки результатов и от того, насколько эти результаты важны. Если нужно быть совершенно уверенными в итогах тестирования, следует прогнать тест большее количество раз. Обычно для оценки применяют такие методики: нахождение лучшего результата, вычисление среднего значения по всем полученным итогам или просто выполнение эталонного теста пять раз и вычисление среднего результата по трем лучшим. Степень точности вы определяете сами. Возможно, вы захотите применить к результатам статистические методы, найти доверительный

[1] Если вам интересно, ни с одним из авторов этого никогда не случалось.

интервал и т. д., но обычно такой уровень точности не требуется[1]. Если тест удовлетворительно отвечает на ваши вопросы, можно просто прогнать его несколько раз и посмотреть, насколько различаются полученные значения. Если различие велико, следует либо увеличить количество прогонов, либо запустить тест на больший период времени, что, как правило, приводит к уменьшению разброса результатов.

После того как результаты получены, необходимо их проанализировать, то есть превратить числа в знания. Цель — получить ответы на вопросы, ради которых готовилась тестовая инфраструктура. Идеально было бы, если бы на выходе формулировались утверждения наподобие «модернизация сервера до четырехпроцессорного увеличит пропускную способность на 50 % при неизменном времени отклика» или «использование индексов ускорило выполнение запросов». Если вы хотите реализовать научный подход к этому вопросу, почитайте о *нулевой гипотезе* до проведения эталонного тестирования, однако имейте в виду, что вряд ли большинство людей ожидают от вас такого высокого уровня.

Методика обработки полученных значений зависит от того, как они были собраны. Вероятно, имеет смысл написать скрипты для анализа результатов — не только для того, чтобы уменьшить объем своей работы, но и по той же причине, по которой следует автоматизировать сам эталонный тест: для обеспечения воспроизводимости и документирования. Ниже представлен очень простой набросок сценария оболочки, который может помочь вам извлечь показатели временных рядов из продемонстрированного нами ранее скрипта, собирающего данные. В качестве параметров командной строки он принимает имена файлов собранных данных:

```
#!/bin/sh

# Этот скрипт переводит SHOW GLOBAL STATUS в формат таблицы,
# каждая строка которой представляет собой одну выборку с измерениями,
# проведенными через промежутки времени, прошедшие между выборками.
awk '
    BEGIN {
       printf "#ts date time load QPS";
       fmt = " %.2f";
    }
    /^TS/ { # The timestamp lines begin with TS.
       ts      = substr($2, 1, index($2, ".") - 1);
       load    = NF - 2;
       diff    = ts - prev_ts;
       prev_ts = ts;
       printf "\n%s %s %s %s", ts, $3, $4, substr($load, 1, length($load)-1);
    }
    /Queries/ {
       printf fmt, ($2-Queries)/diff;
       Queries=$2
    }
    ' "$@"
```

[1] Если вам действительно нужны научно достоверные, скрупулезные результаты, то рекомендуем почитать хорошую книгу по проектированию и выполнению контролируемых тестов, поскольку данная тема намного шире, чем можно изложить здесь.

Если вы назовете скрипт analyze и запустите его для файла статуса, сгенерированного предыдущим скриптом, то можете получить примерно следующее:

```
[baron@ginger ~]$ ./analyze 5-sec-status-2011-03-20
 #ts date time load QPS
 1300642150 2011-03-20 17:29:10 0.00 0.62
 1300642155 2011-03-20 17:29:15 0.00 1311.60
 1300642160 2011-03-20 17:29:20 0.00 1770.60
 1300642165 2011-03-20 17:29:25 0.00 1756.60
 1300642170 2011-03-20 17:29:30 0.00 1752.40
 1300642175 2011-03-20 17:29:35 0.00 1735.00
 1300642180 2011-03-20 17:29:40 0.00 1713.00
 1300642185 2011-03-20 17:29:45 0.00 1788.00
 1300642190 2011-03-20 17:29:50 0.00 1596.40
```

Первая строка содержит заголовки столбцов. На вторую строку не обращайте внимания, поскольку она появляется до запуска эталонного теста. Последующие строки содержат метку времени Unix, дату, время (обратите внимание на то, что данные появляются через пятисекундные интервалы, как говорилось ранее), среднюю загрузку системы и, наконец, количество запросов в секунду (QPS), выполненных сервером базы данных. Это минимальный набор данных, необходимый для изучения производительности системы. Далее покажем, как быстро построить график и посмотреть, что произошло во время эталонного тестирования.

Важность построения графика

Чтобы управлять миром, надо непрерывно строить графики[1]. Но если серьезно, самое простое и полезное, что вы можете сделать с показателями производительности системы, — это представить их в виде временно́го ряда и посмотреть на них.

На графике вы можете сразу выявить проблемы, даже те, которые трудно или невозможно увидеть, исследуя необработанные данные. Вы должны противостоять соблазну просто взглянуть на средние значения и другие сводные статистические данные, которые может выдать инструмент эталонного тестирования. Средние значения бесполезны, потому что скрывают то, что происходит на самом деле. К счастью, вывод, осуществляемый написанными нами скриптами, легко настраивается для таких инструментов, как gnuplot или R, для создания графика в мгновение ока. Мы продемонстрируем использование gnuplot, предположив, что вы сохранили данные в файл под названием `QPS-per-5-seconds`:

```
gnuplot> plot "QPS-per-5-seconds" using 5 w lines title "QPS"
```

Эта строка — указание gnuplot создать в файле пятое поле (поле `QPS`) с линиями, назвать его `QPS` и отобразить на графике. На рис. 2.2 представлен результат.

[1] В оригинале — игра слов. Эту фразу можно перевести так: «Чтобы управлять миром, надо непрерывно плести интриги». — *Примеч. пер.*

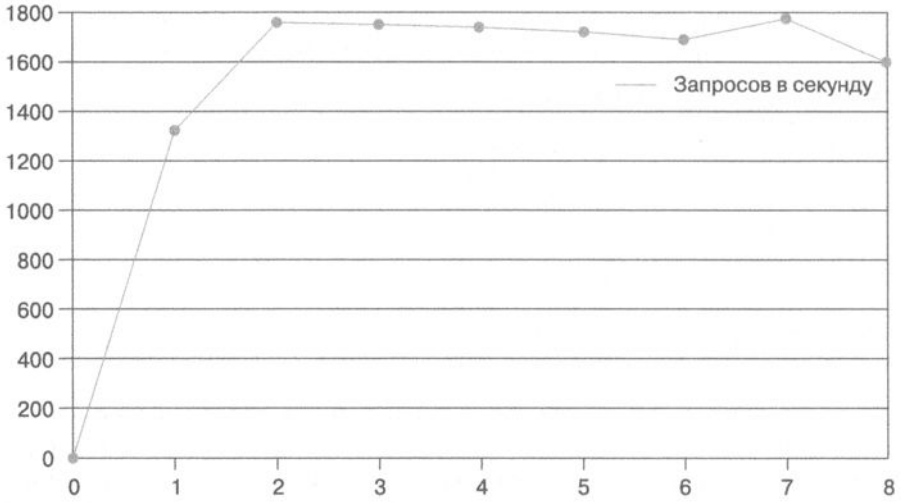


Рис. 2.2. Построение графика по эталонному тесту QPS

Теперь рассмотрим пример, который еще более ярко продемонстрирует важность построения графиков. Предположим, система страдает от так называемого яростного сброса, когда она отстает от проверки и блокирует всю активность до тех пор, пока не нагонит ее, что приводит к резким падениям пропускной способности. Девяносто пятый процентиль и среднее время отклика не покажут падений, поэтому результаты скроют проблему. Однако на графике периодические снижения активности отобразятся (рис. 2.3).

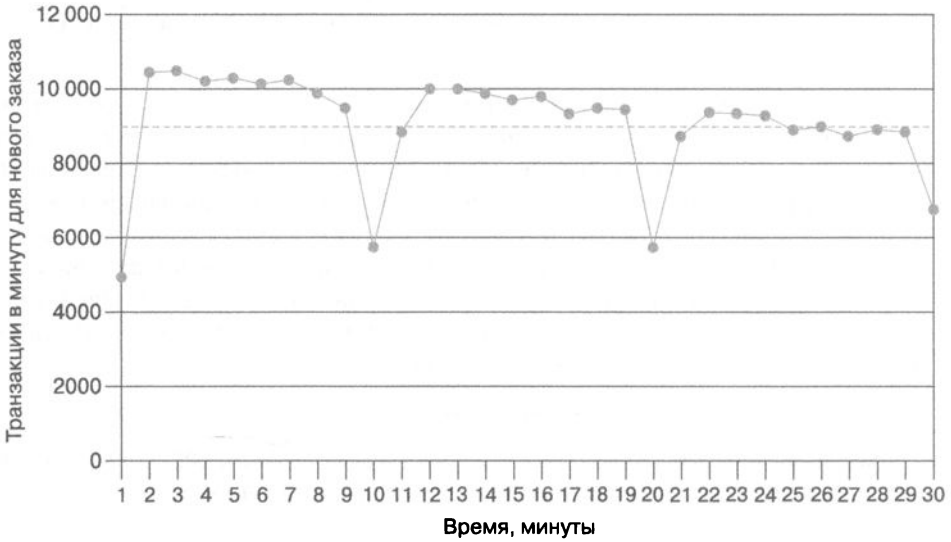


Рис. 2.3. Результаты 30-минутного прогона эталонного теста dbt2

Здесь демонстрируется пропускная способность транзакции в минуту для нового заказа (NOTPM). Эта линия показывает значительные падения, которые график среднего значения (пунктирный) не покажет совсем. Первое падение связано с тем, что кэши сервера еще не прогреты. Другие отражают ситуацию, когда сервер тратит время, интенсивно сбрасывая «грязные» страницы на диск. Без графика трудно обнаружить эти отклонения.

Такие пики часто встречаются в сильно нагруженных системах и требуют изучения. В данном случае такое поведение связано с использованием более старой версии InnoDB, которая имела несовершенный алгоритм сброса. Но нельзя считать это само собой разумеющимся. Следует вернуться к подробной статистике и просмотреть ее. Как выглядел результат `SHOW ENGINE INNODB STATUS` во время этих снижений активности? А как насчет вывода `SHOW FULL PROCESS LIST`? Возможно, вы сразу увидите, что InnoDB выполняла сброс или что в списке процессов было много потоков со статусом «ожидание в очереди из-за блокировки кэша» или подобным. Вот почему полезно очень подробно фиксировать данные во время тестов, а затем строить графики для обнаружения проблем.

Инструменты эталонного тестирования

Если нет веских оснований считать, что готовые инструменты не подходят для ваших целей, то не стоит создавать собственную систему эталонного тестирования. В следующих разделах мы расскажем о некоторых из этих инструментов.

Полностековые инструменты

Вспомним, что существует два типа эталонного тестирования: полностековое и покомпонентное. Нет ничего удивительного в том, что имеются инструменты для тестирования всего приложения и тестирования под нагрузкой MySQL и других компонентов по отдельности. Полностековое тестирование — обычно лучший способ получить полное представление о производительности всей системы. В число инструментов полностекового тестирования входят следующие:

- *ab*. Это инструмент тестирования производительности сервера HTTP Apache. Он показывает, сколько запросов в секунду способен обслуживать HTTP-сервер. Если вы тестируете веб-приложение, это число демонстрирует, какое количество запросов в секунду может обслужить приложение в целом. Это очень простой инструмент, но полезность его ограниченна, поскольку он просто обращается к одному адресу URL настолько быстро, насколько это возможно. Дополнительную информацию об утилите ab вы найдете на странице http://httpd.apache.org/docs/2.0/programs/ab.html;
- *http_load*. Концепция этого инструмента похожа на концепцию ab: он также предназначен для создания нагрузки на веб-сервер, но более гибок. Вы можете создать входной файл, включающий много разных адресов URL, а http_load

будет выбирать их случайным образом. Также можете настроить параметры таким образом, что запросы станут отправляться с заданным интервалом, а не с максимально возможной скоростью. Подробности смотрите на странице http://www.acme.com/software/http_load/;

- *JMeter*. Представляет собой приложение на языке Java, которое может загружать другое приложение и измерять его производительность. Эта утилита была разработана для тестирования веб-приложений, но ее можно использовать и при тестировании FTP-серверов, и при отправке запросов к базе данных через интерфейс JDBC.

  Утилита JMeter значительно сложнее, чем ab и http_load. С ее помощью можно более гибко имитировать поведение реальных пользователей, управляя таким параметром, как время нарастания нагрузки. У нее есть графический пользовательский интерфейс со встроенными средствами построения графиков, также она позволяет сохранять результаты и воспроизводить их в автономном режиме. Подробности смотрите на странице http://jakarta.apache.org/jmeter/.

Инструменты покомпонентного тестирования

Предусмотрены несколько полезных инструментов для тестирования производительности MySQL и операционной системы, на которой она работает. Далее приведем примеры эталонных тестов, в которых эти инструменты используются.

- *mysqlslap* (http://dev.mysql.eom/doc/refman/5.1/en/mysqlslap.html). Имитирует нагрузку на сервер и выдает данные хронометража. Эта утилита является частью дистрибутива MySQL 5.1, но ее можно использовать и с более ранними версиями, начиная с 4.1. Вы можете настроить количество конкурентных соединений и передать программе с помощью либо команды SQL в командной строке, либо файла с командами SQL. Если вы не зададите режим тестирования вручную, программа сама исследует схему базы данных и автоматически сгенерирует команды `SELECT`.
- *MySQL Benchmark Suite (sql-bench)*. Вместе с сервером MySQL распространяется набор инструментов эталонного тестирования, который можно использовать для исследования нескольких различных серверов баз данных. Он однопоточный и измеряет скорость выполнения запросов сервером. Результаты показывают, какие типы операций сервер хорошо выполняет.

  Главным преимуществом этого набора является то, что он содержит множество готовых тестов, так что с его помощью легко сравнивать различные подсистемы хранения или конфигурации. Как высокоуровневый инструмент эталонного тестирования, он полезен для сравнения общей производительности двух серверов. Кроме того, вы можете запускать подмножество тестов (например, тестировать только производительность команды `UPDATE`). В основном тесты нагружают процессоры, но есть короткие периоды, в течение которых выполняется большой объем операций дискового ввода/вывода.

Среди основных недостатков этого инструмента можно выделить следующие: он работает в однопользовательском режиме, задействует очень маленький набор исходных значений, не допускает тестирования на данных, характерных именно для ваших условий, а его результаты могут сильно варьироваться от запуска к запуску. Поскольку он однопоточный и полностью последовательный, с его помощью невозможно оценить выигрыш от наличия нескольких процессоров, но вместе с тем он вполне пригоден для сравнения однопроцессорных серверов.

Эталонное тестирование сервера базы данных требует установки драйверов DBD и языка Perl. Документацию можно найти по адресу http://dev.mysql.com/doc/en/mysql-benchmarks.html/.

- *Super Smack* (http://vegan.net/tony/supersmack/). Это инструмент эталонного тестирования, тестирования под нагрузкой и создания нагрузки для MySQL и PostgreSQL. Он мощный, но довольно сложный, позволяет имитировать работу нескольких пользователей, загружать тестовые данные в базу и заполнять таблицы случайно сгенерированными значениями. Эталонные тесты содержатся в smack-файлах, использующих простой язык определения клиентов, таблиц, запросов и т. д.
- *Database Test Suite.* Инструмент разработан компанией The Open Source Development Labs (OSDL) и размещен на сайте SourceForge по адресу http://sourceforge.net/projects/osdldbt/, представляет собой набор утилит для запуска эталонного тестирования, сходного со стандартными промышленными эталонными тестами, такими как опубликованные Советом по оценке производительности обработки транзакций (Transaction Processing Performance Council, TPC). В частности, инструмент dbt2 представляет собой бесплатную (но несертифицированную) реализацию теста TPC-C OLTP. Мы неоднократно использовали его, пока не разработали специальный инструмент для MySQL.
- *Percona's TPCC-MySQL Tool.* Мы создали реализацию эталонного теста, аналогичного тесту TPC-C, с инструментами, специально разработанными для эталонного тестирования MySQL. Обычно используем его для оценки поведения MySQL при нестандартных нагрузках. (Для более простых эталонных тестов применяем sysbench.) Исходный код доступен на https://launchpad.net/perconatools, также в исходном репозитории хранится краткая документация по использованию.
- *sysbench* (https://launchpad.net/sysbench). Это многопоточный инструмент для эталонного тестирования системы. Цель его применения — получить представление о производительности системы с точки зрения факторов, важных для работы сервера базы данных. Например, вы можете измерить производительность файлового ввода/вывода, планировщика операционной системы, распределения памяти и скорости передачи данных, потоков POSIX и самого сервера базы данных. sysbench поддерживает скрипты на языке Lua (http://www.lua.org), что делает его очень гибким при тестировании множества сценариев. Это наш любимый универсальный инструмент эталонного тестирования MySQL, операционной системы и производительности оборудования.

Функция MySQL BENCHMARK()

В MySQL имеется удобная функция `BENCHMARK()`, которую можно использовать для тестирования скорости выполнения определенных типов операций. Для этого нужно указать количество прогонов и подлежащее выполнению выражение. Им может быть любое скалярное выражение, например скалярный подзапрос или функция. Это удобно для тестирования относительных скоростей некоторых операций, в частности для оценки того, какая из функций, `MD5()` или `SHA1()`, работает быстрее:

```
mysql> SET @input := 'hello world';
mysql> SELECT BENCHMARK(1000000, MD5(@input));
+---------------------------------+
| BENCHMARK(1000000, MD5(@input)) |
+---------------------------------+
|                               0 |
+---------------------------------+
1 row in set (2.78 sec)
mysql> SELECT BENCHMARK(1000000, SHA1(@input));
+----------------------------------+
| BENCHMARK(1000000, SHA1(@input)) |
+----------------------------------+
|                                0 |
+----------------------------------+
1 row in set (3.50 sec)
```

Возвращаемым значением всегда является `0`, клиентское приложение сообщает, сколько времени потребовалось на выполнение запроса. В данном случае, похоже, быстрее работает `MD5()`. Однако корректно использовать функцию `BENCHMARK()` непросто, если вы не совсем понимаете, что она делает. А она просто определяет, как быстро сервер может выполнить выражение, но не сообщает ничего об издержках на синтаксический анализ и оптимизацию. И если выражение не включает в себя пользовательскую переменную, как в нашем примере, то второй и последующие случаи выполнения сервером данного выражения могут оказаться обращением к кэшу.

Несмотря на удобство функции `BENCHMARK()`, мы никогда не применяем ее для реального эталонного тестирования: слишком трудно определить, что она в действительности измеряет. Кроме того, ее результаты относятся лишь к небольшой части всего процесса выполнения.

Примеры эталонного тестирования

В этом разделе мы приведем примеры реальных эталонных тестов, выполненных с помощью вышеупомянутых инструментов. Мы не можем дать исчерпывающее описание каждого инструмента, но эти примеры помогут вам решить, какие эталонные тесты могут оказаться полезными для ваших целей, и начать их использовать.

http_load

Начнем с простого примера, демонстрирующего работу http_load. Мы будем использовать следующие адреса URL, которые сохранили в файле `urls.txt`:

- http://www.mysqlperformanceblog.com/;
- http://www.mysqlperformanceblog.com/page/2/;
- http://www.mysqlperformanceblog.com/mysql-patches/;
- http://www.mysqlperformanceblog.com/mysql-performance-presentations/;
- http://www.mysqlperformanceblog.com/2006/09/06/slow-query-log-analyzes-tools/.

В простейшем варианте http_load извлекает страницы по указанным адресам в цикле. Программа делает это с максимально возможной скоростью:

```
$ http_load -parallel 1 -seconds 10 urls.txt
19 fetches, 1 max parallel, 837929 bytes, in 10.0003 seconds
44101.5 mean bytes/connectior
1.89995 fetches/sec, 83790.7 bytes/sec
msecs/connect: 41.6647 mean, 56.156 max, 38.21 min
msecs/first-response: 320.207 mean, 508.958 max, 179.308 min
HTTP response codes:
  code 200 - 19
```

Результаты вполне понятны — они всего лишь показывают статистическую информацию о запросах.

Немного более сложный сценарий заключается в извлечении адресов URL с максимально возможной скоростью в цикле, но с имитацией пяти параллельных пользователей:

```
$ http_load -parallel 5 -seconds 10 urls.txt
94 fetches, 5 max parallel, 4.75565e+06 bytes, in 10.0005 seconds
50592 mean bytes/connection
9.39953 fetches/sec, 475541 bytes/sec
msecs/connect: 65.1983 mean, 169.991 max, 38.189 min
msecs/first-response: 245.014 mean, 993.059 max, 99.646 min
HTTP response codes:
  code 200 -- 94
```

Вместо максимально быстрого извлечения мы можем симулировать нагрузку для прогнозируемой частоты запросов (например, пять раз в секунду):

```
$ http_load -rate 5 -seconds 10 urls.txt
48 fetches, 4 max parallel, 2.50104e+06 bytes, in 10 seconds
52105 mean bytes/connection
4.8 fetches/sec, 250104 bytes/sec
msecs/connect: 42.5931 mean, 60.462 max, 38.117 min
msecs/first-response: 246.811 mean, 546.203 max, 108.363 min
HTTP response codes:
  code 200 - 48
```

Наконец, имитируем еще большую нагрузку с частотой поступления запросов 20 раз в секунду. Обратите внимание, как с возрастанием нагрузки увеличивается время соединения и отклика:

```
$ http_load -rate 20 -seconds 10 urls.txt
111 fetches, 89 max parallel, 5.91142e+06 bytes, in 10.0001 seconds
53256.1 mean bytes/connection
11.0998 fetches/sec, 591134 bytes/sec
msecs/connect: 100.384 mean, 211.885 max, 38.214 min
msecs/first-response: 2163.51 mean, 7862.77 max, 933.708 min
HTTP response codes:
  code 200 - 11
```

MySQL Benchmark Suite

MySQL Benchmark Suite состоит из набора эталонных тестов, написанных на языке Perl, так что для их запуска вам потребуется Perl. Эталонные тесты находятся в подкаталоге `sql-bench/` установочного каталога MySQL. В системах Debian GNU/Linux, например, они хранятся в каталоге `/usr/ share/mysql/sql-bench/`.

Прежде чем приступать к работе, прочитайте файл `README`, в котором объясняется, как использовать этот комплект инструментов, и описываются аргументы командной строки. Для запуска любых тестов применяются примерно такие команды:

```
$ cd /usr/share/mysql/sql-bench/
sql-bench$ ./run-all-tests --server=mysql --user=root --log --fast
Test finished. You can find the result in:
output/RUN-mysql_fast-Linux_2.4.18_686_smp_i686
```

Эталонное тестирование длится довольно долго — возможно, больше часа в зависимости от оборудования и конфигурации. Если вы укажете параметр командной строки `--log`, то сможете отслеживать состояние теста во время его выполнения. Каждый тест записывает свои результаты в подкаталог `output`. Все файлы содержат последовательности временных меток, соответствующих операциям, реализуемым в каждом эталонном тесте. Вот немного переформатированный для удобства печати образец:

```
sql-bench$ tail -5 output/select-mysql_fast-Linux_2.4.18_686_smp_i686
Time for count_distinct_group_on_key (1000:6000):
   34  wallclock secs ( 0.20 usr  0.08 sys +  0.00 cusr  0.00 csys =  0.28 CPU)
Time for count_distinct_group_on_key_parts (1000:100000):
   34  wallclock secs ( 0.57 usr  0.27 sys +  0.00 cusr  0.00 csys =  0.84 CPU)
Time for count_distinct_group (1000:100000):
   34  wallclock secs ( 0.59 usr  0.20 sys +  0.00 cusr  0.00 csys =  0.79 CPU)
Time for count_distinct_big (100:1000000):
   8   wallclock secs ( 4.22 usr  2.20 sys +  0.00 cusr  0.00 csys =  6.42 CPU)
Total time:
   868 wallclock secs (33.24 usr  9.55 sys +  0.00 cusr  0.00 csys = 42.79 CPU)
```

Например, выполнение теста `count_distinct_group_on_key (1000:6000)` заняло 34 секунды. Это общее время, которое потребовалось клиенту. Другие значения (`usr`, `sys`, `cursr`, `csys`), которые добавили до 0,28 секунды, составляют издержки для

этого теста. Данное значение показывает, сколько времени клиентский код реально работал, а не ожидал ответа от сервера MySQL. Таким образом, интересующее нас значение — время, потраченное на неконтролируемую клиентом обработку, — составило 33,72 секунды.

Тесты можно запускать и по отдельности, а не в составе комплекта. Например, вы можете решить сфокусироваться только на тестировании операций вставки. Полученная информация будет более подробной, чем в сводном отчете, созданном при запуске полного набора тестов:

```
sql-bench$ ./test-insert
Testing server 'MySQL 4.0.13 log' at 2003-05-18 11:02:39

Testing the speed of inserting data into 1 table and do some selects on it.
The tests are done with a table that has 100000 rows.

Generating random keys
Creating tables
Inserting 100000 rows in order
Inserting 100000 rows in reverse order
Inserting 100000 rows in random order
Time for insert (300000):
  42 wallclock secs ( 7.91 usr  5.03 sys +  0.00 cusr  0.00 csys = 12.94 CPU)
Testing insert of duplicates
Time for insert_duplicates (100000):
  16 wallclock secs ( 2.28 usr  1.89 sys +  0.00 cusr  0.00 csys =  4.17 CPU)
```

sysbench

С помощью инструмента sysbench можно запускать различные эталонные тесты. Он разработан не только для тестирования производительности базы данных, но и для определения того, насколько хорошо работает система как сервер баз данных. Фактически Петр и Вадим изначально разработали его для запуска эталонных тестов, особенно важных для оценки производительности MySQL, даже несмотря на то, что они на самом деле не являются эталонными тестами MySQL. Мы начнем с некоторых тестов, не характерных для MySQL и измеряющих производительность подсистем, которые определяют общие ограничения системы. Затем покажем, как измерять производительность базы данных.

Мы настоятельно рекомендуем познакомиться с sysbench. Это один из самых полезных инструментов в наборе пользователя MySQL. И хотя существует множество других инструментов, которые выполняют часть его функций, эти инструменты не всегда надежны и полученные результаты не всегда отражают производительность MySQL. Например, вы можете тестировать производительность ввода/вывода с помощью iozone, bonnie++ и ряда других инструментов, однако потребуется приложить много усилий, чтобы заставить их тестировать ввод/вывод таким же образом, как InnoDB нагружает диски. А вот sysbench ведет себя так же, как InnoDB, поэтому его тест fileio вполне релевантен в поставляемом виде.

Эталонное тестирование процессора с помощью инструмента sysbench

Стандартным тестом подсистемы является эталонный тест процессора, в котором для вычисления простых чисел до указанного максимума используются 64-разрядные целые числа. Мы запустили этот тест на двух серверах, каждый под управлением GNU/Linux, и сравнили результаты. Вот характеристики оборудования первого сервера:

```
[server1 ~]$ cat /proc/cpuinfo
...
model name      : AMD Opteron(tm) Processor 246
stepping        : 1
cpu MHz         : 1992.857
cache size      : 1024 KB
```

И вот как прошел запуск эталонного теста:

```
[server1 ~]$ sysbench --test=cpu --cpu-max-prime=20000 run
sysbench v0.4.8:  multithreaded system evaluation benchmark
...
Test execution summary:    total time:                          121.7404s
```

У второго сервера другой процессор:

```
[server2 ~]$ cat /proc/cpuinfo
...
model name      : Intel(R) Xeon(R) CPU          5130  @ 2.00GHz
stepping        : 6
cpu MHz         : 1995.005
```

Вот результат его эталонного теста:

```
[server1 ~]$ sysbench --test=cpu --cpu-max-prime=20000 run
sysbench v0.4.8:  multithreaded system evaluation benchmark
...
Test execution summary:    total time:                           61.8596s
```

Результат показывает общее время, требуемое для вычисления простых чисел, которое очень легко сравнить. В данном случае второй сервер выполнил эталонный тест приблизительно в два раза быстрее, чем первый.

Эталонное тестирование файлового ввода/вывода с помощью инструмента sysbench

Эталонный тест fileio измеряет параметры работы системы при различных нагрузках ввода/вывода. Он очень полезен для сравнения жестких дисков, RAID-контроллеров и режимов RAID, а также настройки подсистемы ввода/вывода. Тест эмулирует работу InnoDB с дисками.

Первым делом нужно подготовить несколько файлов. Вам необходимо сгенерировать намного больше данных, чем помещается в памяти. Когда данные загрузятся

в память, операционная система будет кэшировать бóльшую их часть, а результаты не совсем точно отразят нагрузку на подсистему ввода/вывода. Начнем с создания набора данных:

```
$ sysbench --test=fileio --file-total-size=150G prepare
```

Эта команда создает файлы в текущем рабочем каталоге, на этапе запуска в него будет производиться запись и из него — чтение.

Вторым шагом является запуск теста. Для тестирования производительности при различных типах нагрузки на систему ввода/вывода используется несколько параметров:

- `seqwr` — последовательная запись;
- `seqrewr` — последовательная перезапись;
- `seqrd` — последовательное чтение;
- `rndrd` — произвольное чтение;
- `rndwr` — произвольная запись;
- `rndrw` — произвольное чтение в сочетании с произвольной записью.

Следующая команда запускает тест ввода/вывода с произвольными чтением/записью:

```
$ sysbench --test=fileio --file-total-size=150G --file-test-mode=rndrw/
--init-rng=on --max-time=300 --max-requests=0 run
```

И вот результаты:

```
sysbench v0.4.8:  multithreaded system evaluation benchmark

Running the test with following options:
Number of threads: 1
Initializing random number generator from timer.

Extra file open flags: 0
128 files, 1.1719Gb each
150Gb total file size
Block size 16Kb
Number of random requests for random IO: 10000
Read/Write ratio for combined random IO test: 1.50
Periodic FSYNC enabled, calling fsync() each 100 requests.
Calling fsync() at the end of test, Enabled.
Using synchronous I/O mode
Doing random r/w test
Threads started!
Time limit exceeded, exiting...
Done.

Operations performed:  40260 Read, 26840 Write, 85785 Other = 152885 Total
Read 629.06Mb  Written 419.38Mb  Total transferred 1.0239Gb  (3.4948Mb/sec)
  223.67 Requests/sec executed

Test execution summary:
    total time:                          300.0004s
    total number of events:              67100
```

```
    total time taken by event execution: 254.4601
    per-request statistics:
         min:                                  0.0000s
         avg:                                  0.0038s

         max:                                  0.5628s
         approx. 95 percentile:                0.0099s
Threads fairness:
    events (avg/stddev):           67100.0000/0.00
    execution time (avg/stddev):   254.4601/0.00
```

Получен большой объем информации. Для измерения производительности подсистемы ввода/вывода наиболее интересны количество запросов в секунду и общая пропускная способность — в данном случае 223,67 запроса в секунду и 3,4948 Мбайт/с соответственно. Эти значения дают хорошее представление о производительности диска.

Закончив тестирование, можете запустить очистку, чтобы удалить файлы, созданные программой sysbench для эталонного тестирования:

```
$ sysbench --test=fileio --file-total-size=150G cleanup
```

Эталонное тестирование OLTP-системы с помощью инструмента sysbench

Эталонное тестирование системы OLTP эмулирует рабочую нагрузку, характерную для обработки транзакций. Приведем пример такого тестирования для таблицы, содержащей 1 миллион строк. В первую очередь нужно подготовить эту таблицу:

```
$ sysbench --test=oltp --oltp-table-size=1000000 --mysql-db=test/
--mysql-user=root prepare
sysbench v0.4.8:  multithreaded system evaluation benchmark

No DB drivers specified, using mysql
Creating table 'sbtest'...
Creating 1000000 records in table 'sbtest'...
```

Это все, что нужно сделать для подготовки тестовых данных. Затем на 60 секунд запустим эталонный тест в режиме чтения с восьмью конкурентными потоками:

```
$ sysbench --test=oltp --oltp-table-size=1000000 --mysql-db=test --mysql-user=root/
--max-time=60 --oltp-read-only=on --max-requests=0 --num-threads=8 run
sysbench v0.4.8:  multithreaded system evaluation benchmark

No DB drivers specified, using mysql
WARNING: Preparing of "BEGIN" is unsupported, using emulation (last message repeated
7 times)
Running the test with following options:
Number of threads: 8

Doing OLTP test.
Running mixed OLTP test Doing read-only test
Using Special distribution (12 iterations,  1 pct of values are returned in 75 pct
cases)
```

```
Using "BEGIN" for starting transactions
Using auto_inc on the id column
Threads started!
Time limit exceeded, exiting...

(последнее сообщение повторяется семь раз)
Done.

OLTP test statistics:
    queries performed:
        read:                            179606
        write:                           0
        other:                           25658
        total:                           205264
    transactions:                        12829  (213.07 per sec.)
    deadlocks:                           0      (0.00 per sec.)
     read/write requests:                179606 (2982.92 per sec.)
     other operations:                   25658  (426.13 per sec.)

Test execution summary:
    total time:                          60.2114s
    total number of events:              12829
    total time taken by event execution: 480.2086

     per-request statistics:
          min:                           0.0030s
          avg:                           0.0374s
          max:                           1.9106s
          approx. 95 percentile:         0.1163s

Threads fairness:
    events (avg/stddev):           1603.6250/70.66
    execution time (avg/stddev):   60.0261/0.06
```

Как и ранее, в полученных результатах содержится очень много информации. Наибольший интерес представляют следующие данные:

- счетчик транзакций;
- количество транзакций в секунду;
- временна́я статистика (минимальное, среднее, максимальное время и 95-й процентиль);
- статистика равномерности загрузки потоков, которая показывает, насколько справедливо распределялась между ними имитированная нагрузка.

Приводимый пример применим к версии 4 инструмента sysbench, которая доступна в предварительно созданных двоичных файлах на SourceForge.net. Но если вы скомпилируете sysbench из исходного кода на Launchpad (это легко и просто!), то сможете воспользоваться большим количеством улучшений, внесенных в версию 5. Вы сможете запускать эталонные тесты не для одной таблицы, а для нескольких, через равные интервалы (например, каждые 10 секунд) видеть пропускную способность и время отклика. Эти показатели очень важны для понимания поведения системы.

Другие возможности инструмента sysbench

Инструмент sysbench может запускать другие эталонные тесты системы, которые непосредственно не измеряют производительность сервера базы данных.

- *memory*. Выполняет последовательные операции чтения или записи в памяти.
- *threads*. Проводит эталонное тестирование производительности планировщика потоков. Особенно полезно тестировать поведение планировщика при высоких нагрузках.
- *mutex*. Измеряет производительность работы мьютексов, эмулируя ситуацию, когда все потоки большую часть времени конкурируют, захватывая мьютекс только на короткое время (мьютекс — это структура данных, которая гарантирует взаимоисключающий доступ к некоторому ресурсу, предотвращая возникновение проблем, связанных с конкурентным доступом).
- *seqwr*. Измеряет производительность последовательной записи. Это очень важно для тестирования практических пределов производительности системы. Тест может показать, насколько хорошо работает кэш RAID-контроллера, и предупредить, если результаты оказываются необычными. Например, если отсутствует кэш записи с резервным питанием от батареи, а диск обрабатывает 3000 запросов в секунду, то что-то идет не так и вашим данным определенно угрожает опасность.

Кроме параметра, задающего режим тестирования (`--test`), инструмент sysbench принимает и некоторые другие общие параметры, такие как `--num-threads`, `--max-requests` и `--maxtime`. Подробное описание этих параметров вы найдете в соответствующей технической документации.

Инструмент dbt2 TPC-C из комплекта Database Test Suite

Инструмент dbt2 из комплекта Database Test Suite представляет собой бесплатную реализацию теста TPC-C. TPC-C — это спецификация (опубликована организацией TPC), которая эмулирует нагрузку, характерную для сложной оперативной обработки транзакций. Этот инструмент сообщает результаты, измеряемые в транзакциях в минуту (tpmC), а также стоимость каждой транзакции (Price/tpmC). Итоги сильно зависят от оборудования, поэтому опубликованные результаты эталонного теста TPC-C содержат подробные характеристики серверов, использованных при его проведении.

Тест dbt2 не является настоящим тестом TPC-C. Он не сертифицирован организацией TPC, и его результаты нельзя непосредственно сравнивать с результатами TPC-C. Обратите также внимание на то, что мы (авторы этой книги) создали инструмент, который, на наш взгляд, лучше, чем dbt2, подходит для MySQL (см. следующий раздел).

Давайте посмотрим, как настраивать и запускать эталонный тест dbt2. Мы использовали его версию 0.37, наиболее свежую из подходящих для MySQL (более поздние

версии содержат исправления, которые MySQL поддерживает не полностью). Выполнены следующие шаги.

1. *Подготовка данных.*

   Следующая команда создает данные по десяти хранилищам данных в указанном каталоге. Вся информация по десяти хранилищам занимает на диске около 700 Мбайт. Требуемое место изменяется пропорционально количеству хранилищ данных, так что можете изменить параметр `-w` для создания набора данных нужного вам размера:

```
# src/datagen -w 10 -d /mnt/data/dbt2-w10
warehouses = 10
districts = 10
customers = 3000
items = 100000
orders = 3000
stock = 100000
new_orders = 900

Output directory of data files: /mnt/data/dbt2-w10

Generating data files for 10 warehouse(s)...
Generating item table data...
Finished item table data...
Generating warehouse table data...
Finished warehouse table data...
Generating stock table data...
```

2. *Загрузка данных в базу MySQL.*

   Следующая команда создает базу данных с названием dbt2w10 и заполняет ее данными, которые мы сгенерировали на предыдущем шаге (флаг `-d` задает имя базы, а `-f` — каталог со сгенерированными данными):

```
# scripts/mysql/mysql_load_db.sh -d dbt2w10 -f /mnt/data/dbt2-w10/
-s /var/lib/mysql/mysql.sock
```

3. *Запуск эталонного теста.*

   Последний шаг заключается в выполнении следующей команды из каталога `scripts`:

```
# run_mysql.sh -c 10 -w 10 -t 300 -n dbt2w10/
-u root -o /var/lib/mysql/mysql.sock-e
************************************************************************
*                     DBT2 test for MySQL  started                     *
*                                                                      *
*            Results can be found in output/9 directory                *
************************************************************************
*                                                                      *
*  Test consists of 4 stages:                                          *
*                                                                      *
*  1. Start of client to create pool of databases connections          *
*  2. Start of driver to emulate terminals and transactions generation *
*  3. Test                                                             *
*  4. Processing of results                                            *
*                                                                      *
************************************************************************

DATABASE NAME:                  dbt2w10
DATABASE USER:                  root
```

```
DATABASE SOCKET:              /var/lib/mysql/mysql.sock
DATABASE CONNECTIONS:         10
TERMINAL THREADS:             100
SCALE FACTOR(WARHOUSES):      10
TERMINALS PER WAREHOUSE:      10
DURATION OF TEST(in sec):     300
SLEEPY in (msec)              300
ZERO DELAYS MODE:             1

Stage 1. Starting up client...
Delay for each thread - 300 msec. Will sleep for 4 sec to start 10 database
connections
CLIENT_PID = 12962

Stage 2. Starting up driver...
Delay for each thread - 300 msec. Will sleep for 34 sec to start 100 terminal
threads
All threads has spawned successfuly.

Stage 3. Starting of the test. Duration of the test 300 sec

Stage 4. Processing of results...
Shutdown clients. Send TERM signal to 12962.
 Response Time (s)
 Transaction       %  Average :  90th %   Total  Rollbacks      %
------------   -----  ---------------  ------  ---------  -----
    Delivery    3.53    2.224 :   3.059    1603          0   0.00
    New Order  41.24    0.659 :   1.175   18742        172   0.92
 Order Status   3.86    0.684 :   1.228    1756          0   0.00
      Payment  39.23    0.644 :   1.161   17827          0   0.00
  Stock Level   3.59    0.652 :   1.147    1630          0   0.00

3396.95 new-order transactions per minute (NOTPM)
5.5 minute duration
0 total unknown errors
31 second(s) ramping up
```

Наиболее важная информация содержится в одной из последних строк:

```
3396.95 new-order transactions per minute (NOTPM)
```

Она показывает, сколько транзакций в минуту может выполнить система. Чем больше это значение, тем лучше. (Термин *new-order* — это не специальное определение для типа транзакции, он просто обозначает, что тест имитирует новый заказ в воображаемом интернет-магазине.)

Вы можете менять некоторые параметры для создания различных эталонных тестов.

- `-c`. Количество соединений с базой данных. Изменяя этот параметр, вы эмулируете различные уровни параллелизма и видите, как система масштабируется.
- `-e`. Этот параметр активизирует режим нулевой задержки и означает, что между запросами не будет задержки. Это позволяет выполнить нагрузочное тестирование базы данных. Однако такая ситуация не вполне реалистична, так как живым пользователям перед генерацией очередного запроса требуется некоторое время подумать.

- `-t`. Общая продолжительность эталонного теста. Выбирайте этот параметр тщательно, иначе результаты будут бессмысленными. Слишком малое время для эталонного тестирования производительности ввода/вывода даст неправильные результаты, поскольку у системы не будет достаточно времени для «прогрева» кэша и перехода в нормальный режим работы. Но если вы хотите выполнить эталонное тестирование системы в режиме загрузки процессора, то не стоит устанавливать это значение слишком большим, поскольку набор данных может заметно вырасти и нагрузка ляжет в основном на подсистему ввода/вывода.

Результаты эталонного теста могут дать информацию не только о производительности. Например, если вы увидите слишком много откатов транзакций, то поймете: скорее всего, что-то в системе работает неправильно.

Инструмент Percona's TPCC-MySQL

Рабочая нагрузка, которую генерирует sysbench, отлично подходит для простых тестов и сравнений, однако она не вполне релевантна для реальных приложений. Для этого намного лучше подходит эталонный тест TPC-C. Хотя инструмент dbt2, показанный в предыдущем разделе, представляет собой одну приличную реализацию этого теста, он имеет некоторые недостатки. Это побудило нас создать еще один инструмент, похожий на TCP-C, но более подходящий для запуска множества очень больших эталонных тестов. Код этого инструмента доступен через Launchpad по адресу https://code.launchpad.net/~percona-dev/perconatools/tpcc-mysql, там же находится небольшой файл `README`, в котором объясняется, как создавать и использовать этот инструмент. Это довольно просто. При большом количестве хранилищ данных вы, возможно, задействуете утилиту параллельной загрузки данных, включенную в инструмент, потому что в противном случае потребуется много времени для создания набора данных.

Для того чтобы использовать этот инструмент, необходимо создать структуру базы данных и таблиц, загрузить данные и выполнить эталонный тест. Структура базы данных и таблиц создается простым скриптом SQL, входящим в исходный код, а загрузка данных осуществляется с помощью программы tpcc_load на языке C, которую вам придется скомпилировать. Тест поработает некоторое время и выдаст большой объем информации. (Всегда целесообразно перенаправлять вывод программы в файлы для документирования, но здесь вам особенно нужно это сделать, иначе вы рискуете потерять даже историю прокрутки.) Далее приведен пример установки, создающий небольшой (пять хранилищ данных) набор данных в базе с именем `tpcc5`:

```
$ ./tpcc_load localhost tpcc5 username p4ssword 5
************************************
*** ###easy### TPC-C Data Loader  ***
************************************
<Parameters>
     [server]: localhost
     [port]: 3306
     [DBname]: tpcc5
```

```
      [user]: username
      [pass]: p4ssword
  [warehouse]: 5
TPCC Data Load Started...
Loading Item
.................................................. 5000
.................................................. 10000
.................................................. 15000

[output snipped for brevity]

Loading Orders for D=10, W= 5
.......... 1000
.......... 2000
.......... 3000
Orders Done.

...DATA LOADING COMPLETED SUCCESSFULLY.
```

Далее нужно запустить эталонный тест, для которого требуется программа tpcc_start на языке C. Здесь опять будет много выходных данных, которые следует перенаправить в файл. Ниже приведен очень короткий пример, который запускает пять потоков для пяти хранилищ данных, «прогревается» 30 секунд, а затем в течение 30 секунд проводит эталонное тестирование:

```
$ ./tpcc_start localhost tpcc5 username p4ssword 5 5 30 30
***************************************
*** ###easy### TPC-C Load Generator ***
***************************************
<Parameters>
     [server]: localhost
     [port]: 3306
     [DBname]: tpcc5
       [user]: username
       [pass]: p4ssword
  [warehouse]: 5
 [connection]: 5
     [rampup]: 30 (sec.)
    [measure]: 30 (sec.)

RAMP-UP TIME.(30 sec.)

MEASURING START.

  10, 63(0):0.40, 63(0):0.42, 7(0):0.76, 6(0):2.60, 6(0):0.17
  20, 75(0):0.40, 74(0):0.62, 7(0):0.04, 9(0):2.38, 7(0):0.75
  30, 83(0):0.22, 84(0):0.37, 9(0):0.04, 7(0):1.97, 9(0):0.80

STOPPING THREADS.....

<RT Histogram>

1.New-Order
2.Payment
3.Order-Status
```

```
4.Delivery
5.Stock-Level

<90th Percentile RT (MaxRT)>
   New-Order : 0.37  (1.10)
     Payment : 0.47  (1.24)
Order-Status : 0.06  (0.96)
    Delivery : 2.43  (2.72)
 Stock-Level : 0.75  (0.79)

<Raw Results>
  [0] sc:221  lt:0  rt:0  fl:0
  [1] sc:221  lt:0  rt:0  fl:0
  [2] sc:23  lt:0  rt:0  fl:0
  [3] sc:22  lt:0  rt:0  fl:0
  [4] sc:22  lt:0  rt:0  fl:0
 in 30 sec.

<Raw Results2(sum ver.)>
  [0] sc:221  lt:0  rt:0  fl:0
  [1] sc:221  lt:0  rt:0  fl:0
  [2] sc:23  lt:0  rt:0  fl:0
  [3] sc:22  lt:0  rt:0  fl:0
  [4] sc:22  lt:0  rt:0  fl:0

<Constraint Check> (all must be [OK])
 [transaction percentage]
        Payment: 43.42% (>=43.0%) [OK]
   Order-Status: 4.52% (>= 4.0%) [OK]
       Delivery: 4.32% (>= 4.0%) [OK]
    Stock-Level: 4.32% (>= 4.0%) [OK]
 [response time (at least 90% passed)]
      New-Order: 100.00%  [OK]
        Payment: 100.00%  [OK]
   Order-Status: 100.00%  [OK]
       Delivery: 100.00%  [OK]
    Stock-Level: 100.00%  [OK]

<TpmC>
                442.000 TpmC
```

Последняя строка — это результат эталонного теста: количество транзакций в минуту, достигнутое в процессе его выполнения[1].

Если вы видите результаты, отклоняющиеся от нормы, в строках, непосредственно предшествующих этой (например в строках проверки ограничений), для поиска ошибок можете проверить гистограммы времени отклика и другую выведенную информацию. Естественно, вы должны были использовать скрипты, аналогичные показанным в этой главе. Кроме того, вам нужны детальные диагностические данные, а также информация о производительности сервера во время выполнения эталонных тестов.

---

[1] В демонстрационных целях мы провели этот тест на ноутбуке. Реальные серверы должны работать намного быстрее.

Итоги главы

У каждого, кто использует MySQL, есть причины разобраться в эталонном тестировании. Эталонное тестирование — это не просто практическая деятельность для решения бизнес-задач, но и образовательный процесс. Изучение того, как структурировать задачу таким образом, чтобы эталонный тест мог помочь найти ответ, аналогично широкому спектру проблем — от работы с лексическими задачами до составления уравнений в математическом курсе. Формулировка правильного вопроса, выбор правильного эталонного теста для ответа на него, определение продолжительности и параметров эталонного теста, его запуск, сбор данных и анализ результатов превратят вас в более грамотного пользователя MySQL.

Если вы еще не знаете sysbench, мы рекомендуем хотя бы ознакомиться с ним. Как минимум узнайте, как использовать его эталонные тесты oltp и fileio. Эталонные тесты oltp очень удобны для быстрого сравнения разных систем. В то же время эталонные тесты файловой системы и диска неоценимы для устранения неполадок и изолирования ошибочных компонентов при возникновении проблем с производительностью системы. Мы часто использовали такие эталонные тесты, чтобы доказать: несмотря на утверждения администратора, у SAN действительно был ошибочный диск, а политика кэширования RAID-контроллера на самом деле не была настроена так, как могло показаться, если судить по информации от утилиты. И когда вы проводите эталонное тестирование диска, который, по его информации, способен выполнять 14 000 произвольных чтений в секунду, вы понимаете, что либо сами допустили ошибку, либо в системе серьезные ошибки, либо она неправильно сконфигурирована[1].

Если вы планируете часто проводить эталонное тестирование систем, целесообразно упорядочить этот процесс. Выберите несколько инструментов эталонного тестирования, которые соответствуют вашим потребностям, и хорошо их изучите. Создайте библиотеку скриптов, помогающих вам при настройке тестов, фиксации программного вывода, выводе информации о производительности системы и статусных сообщений, а затем при ее анализе. Освойтесь с утилитой построения графиков, например gnuplot или R, — не тратьте время на электронные таблицы, они слишком громоздки и медленны. Стройте графики своевременно и регулярно, чтобы выявлять проблемы и неудачи в своих эталонных тестах и системах. Ваши глаза быстрее обнаружат аномалию, чем любой скрипт или инструмент.

[1] Один вращающийся накопитель может выполнить только пару сотен операций в секунду, из-за того что на вращение и поиск нужного фрагмента требуется время.

3 Профилирование производительности сервера

Три наиболее распространенных вопроса, связанных с производительностью, которые мы получаем в консалтинговой практике, — оптимально ли работает сервер, почему конкретный запрос не выполняется достаточно быстро, а также как устранить таинственные перебои в работе, которые пользователи обычно называют «стопор», «захламление» и «зависание». Данная глава даст прямые ответы на эти три типа вопросов. Мы перечислим инструменты и методы, которые помогут повысить общую рабочую нагрузку сервера, ускорить выполнение одного запроса, устранить неполадки и решить трудновыявляемые проблемы, вызванные неизвестным фактором и неизвестно в чем проявляющиеся.

Это может показаться непростой задачей, но оказывается, что с помощью простого метода мы можем извлечь из всего массива информации полезную. Этот метод должен сфокусироваться на измерении того, на что сервер тратит время при своей работе, а методика, которая позволяет это сделать, называется *профилированием*. В текущей главе мы расскажем, как измерять показатели работы системы и создавать профили, а также как профилировать весь стек, от приложения до сервера базы данных и до отдельных запросов.

Начинать необходимо с чистого листа, поэтому сразу развеем несколько распространенных заблуждений об эффективности. Возможно, поначалу это трудно будет понять, но вы оставайтесь с нами, и мы объясним все позже на примерах.

Введение в оптимизацию производительности

Попросите десять человек дать определение термина «производительность», и вы, скорее всего, получите десять различных ответов, в которых будут использованы такие термины, как «запросов в секунду», «загрузка процессора», «масштабируемость» и т. д. Обычно это не вызывает проблем, поскольку люди по-разному понимают производительность в зависимости от контекста, однако в этой главе мы будем использовать формальное определение. Оно гласит: производительность измеряется временем,

требующимся для выполнения задачи. Другими словами, *производительность — это время отклика системы*. Это очень важный подход. Мы измеряем производительность задачами и временем, а не ресурсами. Цель сервера базы данных состоит в выполнении SQL-выражений, поэтому интересующие нас задачи — это запросы или выражения, в основном `SELECT`, `UPDATE`, `INSERT` и ряд других[1]. Производительность сервера базы данных определяется временем отклика на запрос, а единица измерения — это время отклика на запрос.

Теперь зададим еще один риторический вопрос: что такое оптимизация? Мы подробнее ответим на него позже, а пока договоримся, что оптимизация производительности — это работа, направленная на максимальное сокращение времени отклика для данной рабочей нагрузки[2].

Мы обнаружили, что многих людей это сбивает с толку. Например, если вы считаете, что оптимизация производительности требует снижения загрузки процессора, то думаете об уменьшении потребления ресурсов. Но это заблуждение. Ресурсы должны потребляться. Иногда для ускорения работы требуется увеличить потребление ресурсов. Мы многократно обновляли старую версию MySQL с использованием древней версии InnoDB и в результате стали свидетелями резкого увеличения загрузки процессора. Об этом чаще всего не стоит беспокоиться. Обычно это означает, что более новая версия InnoDB тратит больше времени на полезную работу и меньше — на борьбу сама с собой. Проверить время отклика на запрос — лучший способ узнать, полезным ли было обновление. Иногда обновление привносит ошибку, например не использует индекс, что также может проявиться в увеличении загрузки процессора.

А если бы вы считали, что оптимизация производительности связана с увеличением количества запросов в секунду, то подумали бы об оптимизации пропускной способности. Увеличение пропускной способности можно рассматривать как побочный эффект оптимизации производительности[3]. Оптимизация запросов позволяет серверу выполнять больше запросов в секунду, поскольку, когда сервер оптимизирован,

---

[1] Мы не разделяем запросы и выражения, DDL и DML и т. д. Если вы отправляете на сервер команду, неважно какую, вам просто нужно, чтобы она была быстро выполнена. Мы обычно используем слово «запрос» как термин для всех отправляемых команд.

[2] Обычно мы избегаем философских дискуссий об оптимизации производительности, но можем дать две рекомендации по поводу дальнейшего чтения. На сайте Percona (http://www.percona.com) есть статья Goal-Driven Performance Optimization, которая представляет собой краткое руководство пользователя. Кроме того, очень полезно прочитать книгу Кэри Миллсапа (Cary Millsap) Optimizing Oracle Performance (издательство O'Reilly). Метод оптимизации производительности Кэри, метод R, является золотым стандартом в мире Oracle.

[3] Иногда производительность определяют с точки зрения пропускной способности. Так можно делать, но это не то определение, которое мы здесь используем. Считаем, что время отклика более полезно, хотя часто пропускную способность легче измерить в эталонных тестах.

для выполнения каждого из них требуется меньше времени. (Единица измерения пропускной способности — это запросы за время, то есть величина, обратная производительности, с точки зрения нашего определения.)

Таким образом, если нашей целью является сокращение времени отклика, то нужно понять, почему серверу требуется определенное количество времени для ответа на запрос, а также уменьшить или устранить любую ненужную работу, которую он выполняет в процессе достижения результата. Другими словами, следует определить, где затрачивается время, и измерить его. Это приводит ко второму важному принципу оптимизации: *вы не можете достоверно оптимизировать то, что не можете измерить*. Поэтому первая задача заключается в том, чтобы с помощью измерений выяснить, где расходуется время.

Мы заметили, что многие люди, пытаясь что-то оптимизировать, тратят большую часть своего времени на изменения и очень мало — на измерения. Мы же, напротив, стремимся провести большую часть своего времени — возможно, свыше 90 %, — измеряя, где именно задерживается отклик. Если мы не получили ответа, то, возможно, не выполнили измерения правильно или в полном объеме. При сборе полных и должных образом подобранных метрик работы сервера проблемы производительности обычно выплывают на поверхность и решение сразу становится очевидным. Однако измерение может оказаться сложной задачей само по себе, и, кроме того, иногда непонятно, что делать с полученными результатами: выяснить с помощью измерений, где затрачивается время, еще не означает понять, почему это происходит.

Мы упомянули о правильно проведенных измерениях, но что это значит? Правильно проведенное измерение — это измерение, которое затрагивает только ту деятельность, которую вы хотите оптимизировать. Как правило, лишнее попадает в измерения в двух случаях:

- измерения начинаются и заканчиваются не вовремя;
- деятельность измеряется в совокупности, а не по интересующим нас фрагментам.

Например, распространенной ошибкой является изучение поведения всего сервера при медленном выполнении запроса. Если запрос выполняется медленно, лучше всего измерить только его, а не весь сервер. И следует выполнять измерение от начала до конца запроса, а не до или после.

Время на выполнение задачи можно разделить на время выполнения и время ожидания. Для сокращения времени выполнения необходимо определить и измерить подзадачи, а затем произвести одно или несколько следующих действий: полностью исключить подзадачи, сделать их реже выполняемыми или более эффективными. Сокращение времени ожидания — наиболее сложная задача, поскольку ожидание может быть вызвано другими действиями системы и, таким образом, является результатом взаимодействия между нашей задачей и другими задачами, которые могут бороться за доступ к таким ресурсам, как диск или процессор. И вам, возможно,

придется использовать разные методики и инструменты в зависимости от того, потрачено время на выполнение или на ожидание.

В предыдущем абзаце сказано, что вам нужно идентифицировать и оптимизировать подзадачи. Но это упрощение. Редкие или короткие подзадачи могут вносить столь незначительный вклад в общее время отклика, что не стоит тратить время на их оптимизацию. Как же определить, какие задачи требуют оптимизации? Для этого и было придумано профилирование.

Как узнать, верны ли измерения

Измерения очень важны, однако не ошибочны ли они? На самом деле *измерения всегда ошибочны*. Результат измерения величины — это не то же самое, что сама величина. Ошибки в результатах могут быть не настолько большими, чтобы это было важным, и тем не менее это ошибки. Поэтому задаваемый вопрос должен звучать так: «Насколько ошибочными являются измерения?» Эта проблема детально рассматривается в других книгах, поэтому мы не будем здесь на ней останавливаться. Просто будьте в курсе, что вы работаете с результатами измерений, а не с фактическими величинами. Очень часто результаты измерений могут оказаться беспорядочными или двусмысленными, что также может привести к неправильным выводам.

Оптимизация с помощью профилирования

Когда вы изучите ориентированный на время отклика метод оптимизации эффективности и попрактикуетесь в его применении, вы снова и снова будете возвращаться к системам профилирования.

Профилирование является основным средством, с помощью которого в ходе измерений можно выяснить и проанализировать, где происходит потребление времени. Профилирование влечет за собой два шага: измерение уже выполненных задач, а также агрегирование результатов и их сортировку таким образом, чтобы важные задачи были в приоритете.

Все инструменты профилирования работают практически одинаково. Когда начинается выполнение задачи, они запускают таймер, а когда оно заканчивается, останавливают таймер и вычитают время начала из времени окончания, чтобы получить время отклика. Большинство инструментов также записывают, какая служба запустила выполнение задачи. Полученные данные могут быть использованы для построения графиков вызовов, но, что более важно для наших целей, схожие задачи могут быть сгруппированы и объединены. Может быть полезно сделать сложный статистический анализ сгруппированных задач, но в любом случае необходимо определить, сколько задач было сгруппировано, и суммировать время их отклика. Это делает *отчет профилирования*. Он состоит из таблицы, в которой каждой задаче соответствует одна строка. В каждой строке отображаются название задачи, количе-

ство выполняемых задач, общее и среднее время выполнения и доля времени работы, которая пришлась на эту задачу. Отчет о профилировании должен быть отсортирован в порядке убывания суммарного затраченного времени.

Чтобы это стало понятнее, посмотрим на реальный профиль рабочей нагрузки всего сервера, который показывает типы запросов, на выполнение которых сервер тратит свое время. Это самое общее представление о времени отклика, другие варианты мы покажем позже. Далее приведен результат работы инструмента pt-query-digest из пакета Percona Toolkit, который является преемником mak-query-digest из пакета Maatkit. Чтобы не отвлекаться на посторонние моменты, мы немного упростили результат, включив в него только первые несколько типов запросов:

```
Rank Response time    Calls R/Call Item
==== ================ ===== ====== =======
   1 11256.3618 68.1% 78069 0.1442 SELECT InvitesNew
   2  2029.4730 12.3% 14415 0.1408 SELECT StatusUpdate
   3  1345.3445  8.1%  3520 0.3822 SHOW STATUS
```

Здесь приведены лишь несколько первых строк профиля, отсортированные в порядке убывания времени, требовавшегося на отклик. При этом данный профиль содержит минимально необходимый набор столбцов. Каждая строка показывает время отклика, долю, приходящуюся на отклик по данной задаче в общем времени работы системы, количество выполненных запросов, среднее время отклика на запрос и абстракцию запроса. Профиль дает понять, насколько затратен каждый из типов запросов как относительно друг друга, так и относительно общих затрат. В данном случае задачи — это запросы, которые, по-видимому, являются наиболее распространенным способом профилирования MySQL.

Фактически мы обсудим два вида профилирования: *профилирование времени выполнения* и *анализ ожидания*. Профилирование времени выполнения показывает, какие задачи потребляют больше всего времени, а анализ ожидания — на каком этапе задачи застревают или блокируются на самое продолжительное время.

Когда задачи выполняются медленно из-за потребления слишком большого объема ресурсов и тратят бóльшую часть времени на работу, они не будут расходовать много времени на ожидание, поэтому анализ ожидания будет полезен. Верно и обратное: когда задачи все время ждут и не потребляют никаких ресурсов, бесполезно измерять, где они проводят время. Если вы не знаете точно, где притаилась проблема, вам, скорее всего, придется проверить и то и другое. Позже мы приведем несколько таких примеров.

В реальности может случиться, что профилирование времени выполнения покажет: задача потратила слишком много времени. Но, углубившись в изучение проблемы, вы обнаружите, что часть так называемого времени выполнения расходуется на ожидание на более низком уровне. Например, наш упрощенный отчет показал, что много времени потребляет команда `SELECT` из таблицы `InvitesNew`, но это время может быть потрачено на более низком уровне в ходе ожидания завершения ввода/вывода.

Прежде чем вы сможете профилировать систему, вам необходимо научиться измерять ее характеристики, а это часто требует оснащения *инструментами*. У системы, оснащенной инструментами, есть точки измерения, где фиксируются данные, а также способы сделать последние доступными для фиксации. Достаточно оснащенные системы встречаются редко. У большинства из них точек измерения немного, да и те обычно обеспечивают только подсчет действий и не позволяют измерить, сколько времени те заняли. Примером такой системы служит MySQL, по крайней мере до версии 5.5, когда первая версия Performance Schema представила несколько временных точек измерения[1]. В версии MySQL 5.1 и предыдущих практически отсутствуют временные точки измерения; большая часть данных о работе сервера, которую можно было получить, имела вид счетчиков `SHOW STATUS`, которые просто подсчитывали количество совершенных действий. Это основная причина, побудившая нас создать Percona Server, который, начиная с версии 5.0, предлагает оснащение инструментами на уровне запросов.

К счастью, несмотря на наличие нашей методики оптимизации производительности, хорошо работающей благодаря отличным инструментам, вы можете получить результат и без ее использования. Часто можно измерять системы извне, а при невозможности этого — делать обоснованные предположения, опираясь на знания о системе и доступную информацию. Однако в данном случае не забывайте, что вы работаете с потенциально неверными данными и никто не гарантирует корректности ваших предположений. Это риск, которому вы обычно подвергаетесь, наблюдая системы, не являющиеся абсолютно прозрачными.

Например, в Percona Server 5.0 журнал медленных запросов может выявить некоторые из наиболее важных причин низкой производительности, такие как ожидание ввода/вывода на диск или блокировки на уровне строк. Если в журнале отображается 9,6 секунды ожидания ввода/вывода на диск для десятисекундного запроса, не имеет смысла выяснять, на что пришлись оставшиеся 4 % времени отклика. Ввод/вывод на диск, безусловно, является самой важной проблемой.

Интерпретация профиля

Профиль сначала отображает важные задачи, но то, что не показано, может быть столь же важным. Обратимся к приведенному ранее профилю. К сожалению, в нем отсутствует много полезной информации, поскольку он показывает только ранги, суммы и средние значения. Вот какой информации не хватает.

- *Полезность запросов.* Отчет не показывает автоматически, какие запросы стоят того, чтобы их оптимизировать. Это возвращает нас к значению оптимизации. Если вы прочитаете книгу Кэри Миллсапа, то получите намного больше полезной информации по этой теме, но мы повторим два важных нюанса. Во-первых, некоторые задачи не стоит оптимизировать, поскольку время их выполнения составляет незначительную долю от общего времени работы. Согласно закону

[1] Performance Schema в MySQL 5.5 не дает детализации на уровне запросов, эта возможность добавлена в MySQL 5.6.

Амдаля, запрос, который потребляет всего 5 % от общего времени отклика, может ускорить общее время работы только на 5 % независимо от того, насколько быстрее вы его выполните. Во-вторых, если оптимизация задачи стоит 1000 долларов, а бизнес в итоге не получит никаких дополнительных денег, это значит, что вы просто деоптимизировали бизнес на 1000 долларов. Таким образом, оптимизацию необходимо прекратить, если стоимость улучшения превышает получаемую выгоду.

- *Выбросы.* Задачи могут нуждаться в оптимизации, даже если они не оказываются в верхней части профиля. Возможно, медленное выполнение некоторой задачи может быть неприемлемо для пользователей, несмотря на то что эта задача выполняется не настолько часто, чтобы занимать существенную долю в общем времени отклика.
- *Неизвестные неизвестные*[1]. Если это возможно, хороший инструмент профилирования покажет вам потерянное время. *Потерянное время* — это количество часов, не учитываемых в задачах измерения.

  Например, если в результате измерений вы получите общее время работы процессора 10 секунд, а составление вашего профиля подзадач требует 9,7 секунд, тогда потерянное время составляет 300 миллисекунд. Это может быть признаком того, что вы не все измеряете, либо может оказаться неизбежным из-за ошибок округления и самих затрат на измерения. Если инструмент показывает потерянное время, следует обратить на него внимание. Возможно, вы упускаете из виду что-то важное. Если отчет его не показывает, вы должны обратить на это внимание и запомнить (или записать), какой информации вам не хватает. В нашем примере профиля потерянное время не показано, однако это просто недостаток использованного инструмента.
- *Скрытые подробности.* Профиль ничего не говорит о распределении времени отклика. Опираться на средние значения опасно, поскольку они скрывают информацию и не являются хорошим индикатором состояния объекта целиком. Петр любит повторять, что информация о средней температуре по больнице не несет смысловой нагрузки[2]. Что бы случилось, если бы элемент № 1 в показанном ранее отчете состоял из двух запросов с односекундным временем отклика и 12 771 запроса с временем отклика в десятки микросекунд? У нас нет никакого способа выяснить, с какой ситуацией мы имеем дело в данном случае. Чтобы принять правильное решение о месте сосредоточения усилий, вам нужна дополнительная информация о 12 773 запросах, упакованных в эту единственную строчку профиля. Особенно полезно иметь более полную информацию о времени отклика, такую как гистограммы, процентили, стандартное отклонение и индекс рассеяния[3].

---

[1] Мы приносим свои извинения Дональду Рамсфелду. Его комментарии были очень проницательными, даже если звучали смешно.

[2] Чтоб мне провалиться! (Это внутренняя шутка. Мы не смогли устоять.)

[3] Коэффициент дисперсии (или индекс рассеяния) рассчитывается как отношение величины рассеяния к среднему значению. В отечественной литературе шире распространен коэффициент вариации, который рассчитывается как отношение стандартного отклонения (то есть корня из дисперсии) к среднему значению. — *Примеч. пер.*

Хорошие инструменты могут помочь, автоматически показывая эту информацию. Фактически утилита pt-query-digest включает многие из этих показателей в свой профиль и в подробный отчет, который она выдает. В нашем примере профиль существенно упрощен для того, чтобы сосредоточиться на самом важном — сортировке задач по убыванию затраченного времени. Далее в этой главе мы приведем примеры более подробного и полезного отчета о профилировании.

Еще один очень важный момент, опущенный нами в примере профиля, — это возможность анализа взаимодействия на более высоком уровне в стеке. Когда мы смотрим только на запросы на сервере, мы на самом деле не можем сопоставить разные запросы и определить, являются ли они частью одного и того же взаимодействия с пользователем. Можно сказать, у нас туннельное видение и мы не можем уменьшить масштаб и профилировать на уровне транзакций или просмотров страниц. Существует несколько способов решения этой проблемы, например тегирование запросов с помощью специальных сообщений, указывающих, откуда они поступили, с дальнейшим агрегированием на этом уровне. Либо можно добавить инструменты и появится возможность профилирования на уровне приложения, о чем мы расскажем в следующем разделе.

Профилирование приложения

Можно профилировать практически все, что потребляет время, в том числе приложение. На самом деле профилирование приложения, как правило, проще, чем профилирование сервера базы данных и гораздо полезнее. И хотя для иллюстрации мы привели пример профиля о запросах сервера MySQL, лучше измерять и профилировать сверху вниз[1]. Тем самым мы можем отслеживать задачи по мере их прохождения через систему от пользователя к серверам и обратно. Часто именно сервер базы данных виноват в потере производительности, но ничуть не реже эта проблема вызвана ошибкой приложения. Появление узких мест может быть вызвано также одной из следующих причин.

- Выполняется обращение к внешним ресурсам, таким как веб-сервисы или поисковые системы.
- Выполняются операции, которые требуют обработки больших объемов данных в приложении, например синтаксический анализ больших файлов XML.
- Выполняются затратные операции в коротких циклах, например наблюдается злоупотребление регулярными выражениями.
- Используются плохо оптимизированные алгоритмы, например алгоритмы поиска элементов в списках.

К счастью, легко выяснить, является ли источником проблемы MySQL. Для этого необходимо использовать инструмент профилирования приложений. (Кроме того, оно с самого начала поможет разработчикам писать эффективный код.)

[1] Далее приведем примеры, в которых, как мы знаем, источник проблем лежит на нижнем уровне, поэтому в них подход сверху вниз не применяется.

Мы рекомендуем включать код профилирования в каждый новый проект, который вы запускаете. Иногда сложно ввести код профилирования в существующее приложение, но в новые приложения его включить легко.

Не замедлит ли профилирование работу серверов?

Да, профилирование замедляет работу приложения. Нет, профилирование ускоряет работу приложения. Стоп, сейчас мы все объясним.

Профилирование и мониторинг увеличивают издержки. Следует ответить на два вопроса: каковы эти издержки и превышает ли выгода затраты?

Многие из тех, кто занимается проектированием и разработкой высокопроизводительных приложений, полагают, что нужно измерять все что возможно и просто принимать стоимость измерений как часть работы приложения. Гуру производительности Oracle Тома Кайта (Tom Kite) спросили, сколько стоит оснащение инструментами Oracle, и он ответил, что оно позволяет повысить производительность не менее чем на 10 %. Мы согласны с такой точкой зрения и полагаем, что в большинстве приложений, которые в противном случае просто не будут измеряться, прирост производительности, вероятно, будет намного больше, чем 10 %. Даже если вы с этим не согласны, стоит предусмотреть хотя бы облегченное профилирование, которое можно выполнять постоянно. Неприятно обнаруживать узкие места в производительности постфактум просто потому, что вы не встроили в систему средства ежедневного отслеживания изменений производительности. А когда столкнетесь с проблемой, данные о предыдущем состоянии приобретут огромную ценность. Кроме того, вы можете использовать данные профилирования, чтобы планировать приобретение оборудования, выделять ресурсы и прогнозировать нагрузку в пиковые периоды или сезоны.

Что мы называем облегченным профилированием? Хронометраж всех запросов SQL и общего времени исполнения скрипта, несомненно, обходится дешево. Кроме того, вам не надо делать его для каждого просмотра страницы. Если у вас приличный трафик, можно просто выполнить профилирование случайной выборки, включая профилирование в конфигурационном файле приложения:

```
<?php
$profiling_enabled = rand(0, 100) > 99;
?>
```

Профилирование только 1 % просмотров вашей страницы поможет найти самые серьезные проблемы. Особенно полезно делать это в коде, предназначенном для промышленной эксплуатации, поскольку так вы обнаружите то, чего нигде больше не увидите.

Несколько лет назад, когда мы писали второе издание этой книги, для популярных языков и фреймворков веб-программирования еще не было хороших готовых инструментов для профилирования приложений в условиях, далеких от тепличных, поэтому мы показали простой, но эффективный способ создавать собственные

инструменты. Сегодня рады сообщить, что существует большое количество отличных инструментов и все, что вам нужно сделать, — открыть их и начать улучшать производительность.

Прежде всего мы хотим рассказать о достоинствах программного обеспечения New Relic. Нам не платят за его рекламу, и мы обычно не поддерживаем конкретные компании или продукты, но это и правда отличный инструмент. Если вы можете использовать его, сделайте это. Наши клиенты, применяющие New Relic, нередко решают свои проблемы без нашего участия, а иногда с его помощью они обнаруживают проблемы, даже если не могут найти решение. New Relic подключается к вашему приложению, профилирует его и отправляет данные в личный кабинет с веб-интерфейсом, что позволяет сократить время отклика и тем самым повысить производительность приложения. В итоге вы совершаете правильные действия, не задумываясь об этом. New Relic обеспечивает инструменты для всей работы пользователя: от браузера до кода приложения, от базы данных до внутренних операций.

Что замечательно в таких средствах, как New Relic, — это то, что они позволяют оснащать ваш код инструментами в процессе использования, а не только при разработке и не только время от времени. Это важный момент, потому что многие средства для профилирования или оснащения инструментами могут быть настолько затратными, что люди боятся запускать их во время работы.

Следует оснащать систему инструментами в эксплуатационной среде, поскольку тогда вы получите такую информацию о производительности своей системы, которую нельзя получить на стадии разработки или обкатки. Если выбранные вами инструменты действительно слишком затратны для постоянного запуска, попробуйте запустить их хотя бы на одном сервере приложений в кластере или оснащайте инструментами только фрагмент выполнения работы, как указано во врезке «Не замедлит ли профилирование работу серверов?».

Оснащение инструментами приложения PHP. Если вы не можете использовать New Relic, есть и другие хорошие варианты. Для PHP, в частности, есть несколько инструментов, которые могут помочь вам профилировать приложение. Один из них — xhprof (http://pecl.php.net/package/xhprof), разработанный Facebook для собственного применения и открытый для публики в 2009 году. Он имеет множество продвинутых функций, но для нас наиболее важным является то, что его легко установить и использовать, он облегченный, создан для масштабирования, поэтому может работать в эксплуатационной среде даже при очень большой системе. Кроме того, он генерирует хороший профиль вызовов функций, отсортированных по использованному времени. Кроме xhprof, существуют инструменты профилирования низкого уровня, такие как xdebug, Valgrind и cachegrind, которые помогут вам проверить код различными способами[1]. Некоторые из этих инструментов не подходят для использования в боевых условиях из-за их излишней детализации и высоких издержек, но могут быть полезны в среде разработки.

[1] В отличие от PHP многие языки программирования имеют встроенную поддержку профилирования. Для Ruby используйте параметр командной строки -r, для Perl — perl -d: DProf и т. д.

Другое средство профилирования PHP, которое мы обсудим, — это разработанный нами инструмент, частично основанный на коде и принципах, которые мы приводили во втором издании. Он называется Instrument-for-php (IfP) и размещен в Google Code по адресу http //code.google.com/p/instrumentation-for-php/. Он не снабжает инструментами PHP так же скрупулезно, как это делает xhprof, зато он тщательнее работает с вызовами базы данных. Это чрезвычайно удобный способ профилирования использования базы данных приложения в тех случаях, если у вас нет свободного доступа к базе данных или управления ею, что встречается довольно часто. IfP — это класс-одиночка, содержащий счетчики и таймеры, поэтому его легко ввести в готовый продукт, не требуя доступа к конфигурации PHP, что опять же является нормой для многих разработчиков.

IfP не профилирует автоматически все ваши функции PHP, а работает только с самыми важными. Например, вам придется вручную запускать и останавливать пользовательские счетчики для определения того, что вы хотите профилировать. Но он автоматически хронометрирует выполнение всей страницы, упрощает оснащение инструментами базы данных и вызовов memcached, поэтому вам не нужно самим запускать и останавливать счетчики для этих важных элементов. Это означает, что вы можете в одно мгновение профилировать три очень важные области: приложение на уровне запросов (просмотров страниц), запросы к базе данных и запросы к кэшу. IfP также экспортирует счетчики и таймеры в среду Apache, так что Apache может записать результаты в журнал. Это легкий и очень упрощенный способ хранения результатов для дальнейшего анализа. IfP не хранит никаких других данных в ваших системах, поэтому нет необходимости в дополнительном участии системного администратора.

Для его использования просто вызовите `start_request()` в начале выполнения страницы. В идеале это должно быть первым, что делает ваше приложение:

```
require_once('Instrumentation.php');
Instrumentation::get_instance()->start_request();
```

Инструмент вызывает функцию закрытия, поэтому в конце работы вам не нужно ничего делать.

IfP автоматически добавляет комментарии к вашим SQL-запросам. Это позволяет довольно гибко проанализировать приложение, просматривая журнал запросов сервера базы данных, а также легко узнать, что в действительности происходит, когда вы смотрите `SHOW PROCESSLIST` и видите некорректный запрос в MySQL. Большинству людей тяжело дается поиск проблемного запроса, особенно если этот запрос был «слеплен» с помощью конкатенации строк и тому подобных методов, так что найти его в исходном коде простым (текстовым) поиском нельзя. Инструмент сообщит, какой хост приложения отправил запрос, даже если вы используете прокси-сервер или балансировщик нагрузки. Он проинформирует, какой пользователь приложения его выполнил, так что вы сможете найти запрос страницы, функцию в исходном коде и номер строки, а также пары «ключ — значение» для всех созданных счетчиков. Приведем пример:

```
-- File: index.php Line: 118 Function: fullCachePage request_id: ABC session_id: XYZ
SELECT * FROM ...
```

Как оснастить инструментами обращения к MySQL, зависит от того, какой интерфейс вы используете для подключения к системе. При использовании объектно-ориентированного интерфейса mysqli необходимо изменить одну строку: замените вызов конструктора mysqli вызовом автоматически оснащенного инструментами конструктора mysqli_x. Этот конструктор является подклассом, предоставляемым IfP, с переписанным инструментарием и запросами. Если вы не используете объектно-ориентированный интерфейс или применяете какой-то другой уровень доступа к базе данных, вам, скорее всего, потребуется немного переписать код. Надеемся, что в вашем коде нет случайных вызовов базы данных, но если это не так, можете использовать интегрированную среду разработки (IDE), например Eclipse, которая поможет легко его реорганизовать. Централизацию кода доступа к базе данных стоит выполнить по многим причинам.

Анализировать результаты очень легко. Инструмент pt-query-digest из набора Percona Toolkit может извлекать встроенные пары «имя — значение» из комментариев запроса, поэтому вы можете просто записывать запросы в файл журнала MySQL и обрабатывать его. Вы также можете задействовать утилиту mod_log_config от Apache для настройки пользовательского ведения журнала с переменными среды, экспортируемыми с помощью IfP, вместе с макросом `%D` для фиксации времени запроса в микросекундах.

Вы можете загрузить журнал Apache в базу данных MySQL командой `LOAD DATA INFILE` и легко проанализировать SQL-запросы. На сайте IfP есть слайд-шоу в формате PDF с примерами того, как реализовать все эти возможности и многое другое, с образцами запросов и аргументами командной строки.

Допустим, вы не хотите добавлять возможность оснащения инструментами в свое приложение или слишком заняты, чтобы этим заниматься. Но поверьте, это намного проще, чем вы думаете. А приложенные усилия многократно окупятся, позволяя сэкономить время и увеличить производительность. Альтернативы для оснащения инструментами не существует. Используйте New Relic, xhprof, IfP или любое другое из большого набора решений для различных языков и сред приложений — не надо изобретать велосипед.

Анализатор запросов Microsoft Enterprise Monitor

Одним из инструментов, который вы можете использовать, является MySQL Enterprise Monitor. Это часть коммерческой подписки на поддержку MySQL от Oracle. Он может фиксировать запросы на ваш сервер либо из библиотек подключений MySQL, либо из прокси (хотя мы не любители использовать прокси). У него очень красивый графический пользовательский интерфейс, который показывает профиль о запросах на сервере и позволяет легко масштабировать определенный временной интервал, например, во время подозрительного всплеска на графике счетчиков состояния. Также вы можете увидеть такую информацию, как планы запросов `EXPLAIN`, что делает его очень полезным инструментом для диагностики и устранения неполадок.

Профилирование запросов MySQL

Существует два подхода к профилированию запросов, соответствующие двум вопросам, упомянутым в начале главы. Можно профилировать весь сервер, основываясь на том, какие запросы в наибольшей степени его загружают. (Если вы начали с верхнего уровня, с профилирования на уровне приложений, то, возможно, уже знаете, какие запросы требуют внимания.) Затем, как только настроите конкретные запросы для оптимизации, можете углубиться в их профилирование по отдельности, определяя, какие подзадачи значительнее увеличивают их время отклика.

Профилирование рабочей нагрузки сервера

Подход к профилированию на уровне сервера очень полезен, потому что он может помочь вам проверить сервер на предмет неэффективных запросов. Обнаружение и исправление этих запросов позволит улучшить производительность приложения в целом, а также выявить конкретные проблемы. Вы можете снизить общую нагрузку на сервер, тем самым уменьшится конкуренция за совместно используемые ресурсы и увеличится скорость выполнения всех запросов (побочный эффект). Снижение нагрузки на сервер способно помочь вам отложить обновления и другие затратные мероприятия или избежать их, кроме того, вы можете обнаружить и устранить проблемы, связанные с результатами, неприемлемыми для пользователя, такими как выбросы (аномальные результаты измерений).

С каждой следующей версией в MySQL появляется все больше инструментов, и если такая тенденция сохранится, скоро в MySQL будут великолепные инструменты для измерения наиболее важных аспектов ее производительности. Но что касается профилирования запросов и поиска самых затратных из них, нам не нужна вся эта сложность. Необходимый инструмент существует уже довольно давно. Это так называемый *журнал медленных запросов.*

Фиксация запросов MySQL в журнал

В MySQL журнал медленных запросов изначально предназначался для фиксации только медленных запросов, но для целей профилирования необходима регистрация *всех* запросов. При этом нам требуется более тонкая детализация времени отклика, чем в MySQL 5.0 и ранних версиях. В этих версиях минимальный временной интервал равнялся 1 секунде. К счастью, прежние ограничения уже неактуальны.

В MySQL 5.1 и более поздних версиях журнал медленных запросов расширен так, что переменную сервера `long_query_time` можно установить равной нулю, зафиксировав все запросы, а время отклика на запрос детализировано с дискретностью 1 микросекунда. Если вы используете Percona Server, этот функционал доступен уже в версии 5.0, кроме того, Percona Server дает намного больший контроль над содержимым журнала и фиксацией запросов.

В существующих версиях MySQL у журнала медленных запросов наименьшие издержки и наибольшая точность измерения времени выполнения запроса. Если вас беспокоит дополнительный ввод/вывод, вызываемый этим журналом, то не тревожьтесь. Мы провели эталонное тестирование и выяснили, что при нагрузках, связанных с вводом/выводом, издержки незначительны. (На самом деле это лучше видно в ходе работ, нагружающих процессор.) Более актуальной проблемой является заполнение диска. Убедитесь, что вы установили смену журнала для журнала медленных запросов, если он включен постоянно. Либо оставьте его выключенным и включайте только на определенное время для получения образца рабочей нагрузки.

У MySQL есть и другой тип журнала запросов — общий журнал, но он не так полезен для анализа и профилирования сервера. Запросы регистрируются по мере их поступления на сервер, поэтому журнал не содержит информации о времени отклика или о плане выполнения запроса. MySQL 5.1 и более поздние версии поддерживают также ведение журнала запросов к таблицам, однако это не самая удачная идея. Данный журнал сильно влияет на производительность: хотя MySQL 5.1 в журнале медленных запросов отмечает время запросов с точностью до 1 микросекунды, медленные запросы к таблице регистрируются с точностью до 1 секунды. Это не очень полезно.

Percona Server регистрирует в журнале медленных запросов значительно более подробную информацию, чем MySQL. Здесь отмечается полезная информация о плане выполнения запроса, блокировке, операциях ввода/вывода и многом другом. Эти дополнительные биты данных добавлялись медленно, поскольку мы столкнулись с различными сценариями оптимизации, которые требовали более подробных сведений о том, как запросы выполняются и где происходят затраты времени. Мы также упростили администрирование. Например, добавили возможность глобально контролировать порог `long_query_time` для каждого соединения, поэтому вы можете заставить их запускать или останавливать журналирование своих запросов, когда приложение использует пул соединений или постоянные соединения, но не можете сбросить переменные уровня сеанса.

В целом это легкий и полнофункциональный способ профилирования сервера и оптимизации его запросов.

Допустим, вы не хотите регистрировать запросы на сервере или по какой-то причине не можете делать этого, например не имеете доступа к серверу. Мы сталкивались с такими ограничениями, поэтому разработали две альтернативные методики и добавили их в инструмент pt-query-digest пакета Percona Toolkit. Первая методика подразумевает постоянное отслеживание состояния с помощью команды `SHOW FULL PROCESSLIST` с параметром `--processlist`. При этом отмечается, когда запросы появляются и исчезают. В некоторых случаях этот метод довольно точен, но он не может зафиксировать все запросы. Очень короткие запросы могут проскочить и завершиться, прежде чем инструмент их заметит.

Второй метод состоит в фиксировании сетевого трафика TCP и его проверки, а затем декодирования протокола «клиент/сервер MySQL» (MySQL client/server protocol). Вы можете использовать утилиту tcpdump для записи трафика на диск,

а затем — pt-query-digest с параметром `--type=tpcdump` для декодирования и анализа запросов. Это гораздо более точная методика, которая зафиксирует все запросы. Методика работает даже с расширенными протоколами, такими как бинарный протокол, используемый для создания и выполнения подготовленных операторов на стороне сервера, и сжатый протокол. Можно также использовать MySQL Proxy со скриптом журналирования, но в практике это нам редко встречалось.

Анализ журнала запросов

Мы рекомендуем по крайней мере время от времени фиксировать в журнале медленных запросов все запросы, выполняемые на сервере, и анализировать их. Запишите запросы, сделанные в течение репрезентативного периода времени, например за час пикового трафика. Если рабочая нагрузка однородна, достаточно будет минуты или даже меньше для нахождения плохих запросов, которые следует оптимизировать.

Не стоит просто открывать журнал и смотреть в него — это пустая трата времени и денег. Сначала создайте профиль и, если это необходимо, просмотрите конкретные выборки в журнале. Лучше всего начинать с верхнего уровня и двигаться вниз, в противном случае, как упоминалось ранее, вы можете деоптимизировать процесс.

Для создания профиля из журнала медленных запросов требуется хороший инструмент анализа журналов. Мы предлагаем утилиту pt-query-digest, которая, возможно, является самым мощным инструментом анализа журнала запросов MySQL. Она поддерживает множество различных функций, включая возможность сохранять отчеты о запросах в базе данных и отслеживать изменения в рабочей нагрузке.

По умолчанию вы просто выполняете утилиту, передаете ей файл журнала медленных запросов в качестве аргумента, и она сама выполняет нужные действия. Утилита выводит профиль запросов в журнале, а затем выбирает важные классы запросов и выдает подробный отчет по каждому из них. В отчете есть десятки мелочей, облегчающих вашу жизнь. Мы продолжаем активно развивать этот инструмент, поэтому лучше прочитать документацию для последней версии, чтобы узнать о ее функциональности.

Приведем краткий обзор отчета pt-query-digest, начиная с профиля. Ниже представлена полная версия профиля, который мы показали ранее в этой главе:

```
# Profile
# Rank Query ID           Response time    Calls R/Call V/M   Item
# ==== ================== ================ ===== ====== ===== =======
#    1 0xBFCF8E3F293F6466 11256.3618 68.1% 78069 0.1442  0.21 SELECT InvitesNew?
#    2 0x620B8CAB2B1C76EC  2029.4730 12.3% 14415 0.1408  0.21 SELECT StatusUpdate?
#    3 0xB90978440CC11CC7  1345.3445  8.1%  3520 0.3822  0.00 SHOW STATUS
#    4 0xCB73D6B5B031B4CF  1341.6432  8.1%  3509 0.3823  0.00 SHOW STATUS
# MISC 0xMISC               560.7556  3.4% 23930 0.0234   0.0 <17 ITEMS>
```

Здесь показано чуть больше деталей, чем раньше. Во-первых, каждый запрос имеет идентификатор, который является хеш-подписью его цифрового отпечатка. Цифровой отпечаток — это нормализованная каноническая версия запроса с уда-

ленными литералами и пробелами, переведенная в нижний регистр (обратите внимание, что запросы 3 и 4 кажутся одинаковыми, но у них разные отпечатки). Инструмент также объединяет таблицы с похожими именами в каноническую форму. Вопросительный знак в конце имени таблицы `InvitesNew` означает, что к имени таблицы был добавлен идентификатор сегмента данных (шарда), а инструмент удалил его, так что запросы к таблицам, сделанные с похожими целями, объединены вместе. Этот отчет взят из сильно шардированного приложения Facebook.

Еще один появившийся здесь столбец — отношение рассеяния к среднему значению `V/M`. Этот показатель называется *индексом рассеяния*. У запросов с более высоким индексом рассеяния сильнее колеблется время выполнения, и они, как правило, являются хорошими кандидатами на оптимизацию. Если вы укажете параметр `--explain` в утилите pt-query-digest, то к таблице будет добавлен столбец с кратким описанием плана запроса `EXPLAIN` — своего рода неформальный код запроса. Это в сочетании со столбцом `V/M` позволяет быстро определить, какие запросы являются плохими и потенциально легко оптимизируемыми.

Наконец, в нижней части есть дополнительная строка, показывающая наличие 17 других типов запросов, которые инструмент не счел достаточно важными для отдельной строки, и сводная статистика по ним. При задании параметров `--limit` и `--outliers` инструмент не будет сворачивать несущественные запросы в одну финальную строку. По умолчанию он выводит запросы, которые входят в первую десятку по затраченному времени либо время выполнения которых превысило односекундный порог во много раз. Оба этих параметра можно настроить.

После профиля инструмент вывел подробный отчет по каждому типу запроса. Вы можете сравнить отчеты по запросам с профилем, сопоставляя по идентификатору запроса или рангу. Приведем отчет для худшего запроса:

```
# Query 1: 24.28 QPS, 3.50x concurrency, ID 0xBFCF8E3F293F6466 at byte 5590079
# This item is included in the report because it matches --limit.
# Scores: V/M = 0.21
# Query_time sparkline: | _^_.^_ |
# Time range: 2008-09-13 21:51:55 to 22:45:30
# Attribute    pct   total     min     max     avg     95%  stddev  median
# ============ === ======= ======= ======= ======= ======= ======= =======
# Count         63   78069
# Exec time     68  11256s    37us      1s   144ms   501ms   175ms    68ms
# Lock time     85    134s       0   650ms     2ms   176us    20ms    57us
# Rows sent      8  70.18k       0       1    0.92    0.99    0.27    0.99
# Rows examine   8  70.84k       0       3    0.93    0.99    0.28    0.99
# Query size    84  10.43M     135     141  140.13  136.99    0.10  136.99
# String:
# Databases    production
# Hosts
# Users        fbappuser
# Query_time distribution
#   1us
#  10us  #
# 100us  ##################################################
#   1ms  ###
```

```
#  10ms  ################
# 100ms  ################################################################
#    1s  #
#  10s+
# Tables
#    SHOW TABLE STATUS FROM `production ` LIKE'InvitesNew82'\G
#    SHOW CREATE TABLE `production `.`InvitesNew82'\G
# EXPLAIN /*!50100 PARTITIONS*/ SELECT InviteId, InviterIdentifier FROM InvitesNew82
WHERE (InviteSetId = 87041469) AND (InviteeIdentifier = 1138714082) LIMIT 1\G
```

Вверху отчета содержится множество метаданных, в том числе: как часто выполняется запрос, его средняя конкурентность и смещение (в байтах) до того места, где в файле журнала находится запрос с наихудшей производительностью. Существует табличная распечатка числовых метаданных, включая статистику, такую как, например, стандартное отклонение[1].

Затем представлена гистограмма времени отклика. Любопытно, что, как вы видите под строкой `Query_time distribution`, у гистограммы этого запроса два пика. Обычно запрос выполняется за сотни миллисекунд, но есть также значительный всплеск числа запросов, которые были выполнены на три порядка быстрее. Если бы этот журнал был создан в пакете Percona Server, в журнале запросов был бы более богатый набор параметров. Как следствие, мы могли бы проанализировать запросы вдоль и поперек, чтобы понять, почему это происходит. Возможно, это были запросы к определенным значениям, которые непропорционально распределены, поэтому использовался другой индекс, или, возможно, это хиты запросов кэша. В реальных системах гистограмма с двумя пиками не редкость, особенно в случае простых запросов, которые часто имеют лишь несколько альтернативных путей выполнения.

Наконец, раздел деталей отчета заканчивается небольшими вспомогательными фрагментами для облегчения копирования и вставки команд в командную строку, а также проверки схемы и статуса упомянутых таблиц и включает образец запроса `EXPLAIN`. Образец содержит все литералы, а не «отпечатки пальцев», поэтому это реальный запрос. На самом деле это экземпляр запроса, у которого было худшее время выполнения в нашем примере.

После выбора запросов, которые вы хотите оптимизировать, можете использовать этот отчет, чтобы быстро проверить выполнение запроса. Мы постоянно пользуемся этим инструментом, и потратили много времени на то, чтобы сделать его максимально эффективным и полезным. Настоятельно рекомендуем подружиться с ним. Возможно, в скором времени MySQL будет лучше оснащена встроенными инструментами профилирования, но на момент написания книги нет инструментов лучше, чем журналирование запросов с помощью журнала медленных запросов или использование tcpdump и запуск полученного журнала с помощью утилиты pt-query-digest.

---

[1] Для ясности мы упрощаем, но журнал запросов Percona Server дает намного более подробный отчет, который поможет вам понять, почему запрос тратит 144 миллисекунды для изучения одной строки — это много!

Профилирование отдельных запросов

После того как вы определили запрос для оптимизации, можете углубиться в него и определить, почему он требует столько времени и как его оптимизировать. Современные методики оптимизации запросов описаны в последующих главах этой книги вместе с необходимыми для их использования предварительными сведениями. Сейчас наша цель — просто показать, как измерить то, что делает запрос, и сколько времени требует каждый этап. Эти знания помогут вам определить, какие методики оптимизации использовать.

К сожалению, большинство инструментов в MySQL не очень полезны для профилирования запросов. Ситуация меняется, но на момент написания книги большинство производственных серверов не поддерживают новейших функций профилирования. Поэтому при их использовании в практических целях мы сильно ограничены командами `SHOW STATUS`, `SHOW PROFILE` и изучением отдельных записей в журнале медленных запросов (если у вас есть Percona Server — в стандартной системе MySQL в журнале нет дополнительной информации). Мы продемонстрируем все три метода на примере одного и того же запроса и покажем, что вы можете узнать о его выполнении в каждом случае.

Команда SHOW PROFILE

Команда `SHOW PROFILE` появилась благодаря Джереми Коулу (Jeremy Cole). Она включена в MySQL 5.1 и более поздние версии. Это единственный реальный инструмент профилирования запросов, доступный в GA-релизе MySQL на момент написания книги. Профилирование по умолчанию отключено, но его можно включить во время сеанса, установив значение переменной сервера:

```
mysql> SET profiling = 1;
```

После этого всякий раз, когда вы посылаете выражение на сервер, он будет замерять прошедшее время и еще некоторые данные, когда запрос будет переходить из одного состояния выполнения в другое. Эта команда имеет довольно широкий функционал и была спроектирована так, что может иметь еще больше, но в следующих релизах она, по всей видимости, будет заменена или вытеснена Performance Schema. Несмотря на это, наиболее полезной функцией данной команды является создание профиля работы, выполняемой сервером во время реализации выражения.

Каждый раз, когда вы отправляете запрос на сервер, он записывает информацию профилирования во временную таблицу и присваивает выражению целочисленный идентификатор, начиная с 1. Приведем пример профилирования представления, включенного в базу данных Sakila[1]:

```
mysql> SELECT * FROM sakila.nicer_but_slower_film_list;
[query results omitted]
997 rows in set (0.17 sec)
```

[1] Представление слишком большое, чтобы привести его здесь, но базу данных Sakila можно скачать с сайта MySQL.

Запрос возвратил 997 строк примерно через 1/6 секунды. Посмотрим, что выдаст команда `SHOW PROFILES` (обратите внимание на множественное число):

```
mysql> SHOW PROFILES;
+----------+------------+-----------------------------------------------+
| Query_ID | Duration   | Query                                         |
+----------+------------+-----------------------------------------------+
|        1 | 0.16767900 | SELECT * FROM sakila.nicer_but_slower_film_list |
+----------+------------+-----------------------------------------------+
```

Первое, что мы видим, — то, что время отклика запроса показано с большей точностью, чем принято. Двух десятичных знаков точности, как показано в клиенте MySQL, часто недостаточно, когда вы работаете с быстрыми запросами. Теперь посмотрим на профиль для этого запроса:

```
mysql> SHOW PROFILE FOR QUERY 1;
+----------------------+----------+
| Status               | Duration |
+----------------------+----------+
| starting             | 0.000082 |
| Opening tables       | 0.000459 |
| System lock          | 0.000010 |
| Table lock           | 0.000020 |
| checking permissions | 0.000005 |
| checking permissions | 0.000004 |
| checking permissions | 0.000003 |
| checking permissions | 0.000004 |
| checking permissions | 0.000560 |
| optimizing           | 0.000054 |
| statistics           | 0.000174 |
| preparing            | 0.000059 |
| Creating tmp table   | 0.000463 |
| executing            | 0.000006 |
| Copying to tmp table | 0.090623 |
| Sorting result       | 0.011555 |
| Sending data         | 0.045931 |
| removing tmp table   | 0.004782 |
| Sending data         | 0.000011 |
| init                 | 0.000022 |
| optimizing           | 0.000005 |
| statistics           | 0.000013 |
| preparing            | 0.000008 |
| executing            | 0.000004 |
| Sending data         | 0.010832 |
| end                  | 0.000008 |
| query end            | 0.000003 |
| freeing items        | 0.000017 |
| removing tmp table   | 0.000010 |
| freeing items        | 0.000042 |
| removing tmp table   | 0.001098 |
| closing tables       | 0.000013 |
| logging slow query   | 0.000003 |
| logging slow query   | 0.000789 |
| cleaning up          | 0.000007 |
+----------------------+----------+
```

Профиль позволяет следить за каждым шагом выполнения запроса и видеть, сколько прошло времени. Обратите внимание, что не очень легко просмотреть выведенный результат и найти, где затраты времени были максимальными: он сортируется в хронологическом порядке. Однако нас интересует не порядок, в котором выполнялись операции, — мы просто хотим знать, каковы были затраты времени на них. К сожалению, отсортировать вывод с помощью `ORDER BY` нельзя. Давайте перейдем к использованию команды `SHOW PROFILE` для запроса связанной таблицы `INFORMATION_SCHEMA` и формата, который выглядит как просмотренные нами ранее профили:

```
mysql> SET @query_id = 1;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT STATE, SUM(DURATION) AS Total_R,
->     ROUND(
->        100 * SUM(DURATION) /
->           (SELECT SUM(DURATION)
->            FROM INFORMATION_SCHEMA.PROFILING
->            WHERE QUERY_ID = @query_id
->        ), 2) AS Pct_R,
->     COUNT(*) AS Calls,
->     SUM(DURATION) / COUNT(*) AS "R/Call"
-> FROM INFORMATION_SCHEMA.PROFILING
-> WHERE QUERY_ID = @query_id
-> GROUP BY STATE
-> ORDER BY Total_R DESC;
+----------------------+----------+-------+-------+--------------+
| STATE                | Total_R  | Pct_R | Calls | R/Call       |
+----------------------+----------+-------+-------+--------------+
| Copying to tmp table | 0.090623 | 54.05 |     1 | 0.0906230000 |
| Sending data         | 0.056774 | 33.86 |     3 | 0.0189246667 |
| Sorting result       | 0.011555 |  6.89 |     1 | 0.0115550000 |
| removing tmp table   | 0.005890 |  3.51 |     3 | 0.0019633333 |
| logging slow query   | 0.000792 |  0.47 |     2 | 0.0003960000 |
| checking permissions | 0.000576 |  0.34 |     5 | 0.0001152000 |
| Creating tmp table   | 0.000463 |  0.28 |     1 | 0.0004630000 |
| Opening tables       | 0.000459 |  0.27 |     1 | 0.0004590000 |
| statistics           | 0.000187 |  0.11 |     2 | 0.0000935000 |
| starting             | 0.000082 |  0.05 |     1 | 0.0000820000 |
| preparing            | 0.000067 |  0.04 |     2 | 0.0000335000 |
| freeing items        | 0.000059 |  0.04 |     2 | 0.0000295000 |
| optimizing           | 0.000059 |  0.04 |     2 | 0.0000295000 |
| init                 | 0.000022 |  0.01 |     1 | 0.0000220000 |
| Table lock           | 0.000020 |  0.01 |     1 | 0.0000200000 |
| closing tables       | 0.000013 |  0.01 |     1 | 0.0000130000 |
| System lock          | 0.000010 |  0.01 |     1 | 0.0000100000 |
| executing            | 0.000010 |  0.01 |     2 | 0.0000050000 |
| end                  | 0.000008 |  0.00 |     1 | 0.0000080000 |
| cleaning up          | 0.000007 |  0.00 |     1 | 0.0000070000 |
| query end            | 0.000003 |  0.00 |     1 | 0.0000030000 |
+----------------------+----------+-------+-------+--------------+
```

Так намного лучше! Теперь мы видим, что причина, по которой этот запрос так долго выполнялся, заключалась в копировании данных во временную таблицу, на что затрачено более половины общего времени. Возможно, придется переписать

этот запрос так, чтобы он не использовал временную таблицу или хотя бы делал это более эффективно. Следующий крупный потребитель времени, отправка данных, фактически является своеобразной кладовкой, которая может включать в себя любое количество различных действий сервера, в том числе поиск совпадающих строк в соединении и т. д. Трудно сказать, сможем ли мы здесь что-то сэкономить. Обратите внимание на то, что результат сортировки занимает недостаточно много времени, чтобы его можно было оптимизировать. Это довольно типично, поэтому призываем вас не тратить время на «настройку буферов сортировки» и подобные мероприятия.

Как обычно, хотя профиль помогает нам определить, какие виды деятельности вносят наибольший вклад в затраченное на выполнение запроса время, он не говорит нам, *почему* это происходит. Чтобы узнать, почему потребовалось столько времени для копирования данных во временную таблицу, нам пришлось бы углубиться в этот процесс и создать профиль выполняемых подзадач.

Команда SHOW STATUS

Команда `SHOW STATUS` MySQL возвращает множество счетчиков. Существует глобальная область действия сервера для счетчиков, а также область сеанса, которая специфична для конкретного соединения. Например, счетчик `Queries` в начале вашего сеанса равен нулю и увеличивается каждый раз, когда вы делаете запрос. Выполнив команду `SHOW GLOBAL STATUS` (обратите внимание на добавление ключевого слова `GLOBAL`), вы увидите общее количество запросов, полученных с момента его запуска. Области видимости разных счетчиков различаются — счетчики, которые не имеют области видимости на уровне сеанса, отображаются в `SHOW STATUS`, маскируясь под счетчики сеансов, и это может ввести в заблуждение. Учитывайте это при использовании данной команды. Как говорилось ранее, подбор должным образом откалиброванных инструментов является ключевым фактором успеха. Если вы пытаетесь оптимизировать что-то, что можете наблюдать только в конкретном соединении с сервером, измерения, которые «засоряются» всей активностью сервера, вам не помогут. В руководстве по MySQL есть отличное описание всех переменных, имеющих как глобальную, так и сеансовую область видимости.

`SHOW STATUS` может быть полезным инструментом, но на самом деле его применение — это не профилирование[1]. Большинство результатов команды `SHOW STATUS` — всего лишь счетчики. Они сообщают вам, как часто протекали те или иные виды деятельности, например чтение из индекса, но ничего не говорят о том, сколько времени на это было затрачено. В команде `SHOW STATUS` есть только один счетчик, который показывает время, израсходованное на операцию (`Innodb_row_lock_time`), но он имеет лишь глобальную область видимости, поэтому вы не можете использовать его для проверки работы, выполненной в ходе сеанса.

[1] Если у вас есть второе издание нашей книги, вы можете заметить, что наше мнение по этому вопросу кардинально изменилось.

Хотя команда SHOW STATUS не предоставляет информацию о затратах времени, тем не менее иногда может быть полезно использовать ее после выполнения запроса для просмотра значений некоторых счетчиков. Вы можете сделать предположение о том, какие типы затратных операций выполнялись и как они могли повлиять на время запроса. Наиболее важными счетчиками являются счетчики обработчиков запросов и счетчики временных файлов и таблиц. Более подробно рассмотрим их в приложении Б. А сейчас приведем пример сброса счетчиков состояния сеанса до нуля, выбора из использованного нами ранее представления и просмотра счетчиков:

```
mysql> FLUSH STATUS;
mysql> SELECT * FROM sakila.nicer_but_slower_film_list;
[query results omitted]
mysql> SHOW STATUS WHERE Variable_name LIKE 'Handler%'
          OR Variable_name LIKE 'Created%';

+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| Created_tmp_disk_tables    | 2     |
| Created_tmp_files          | 0     |
| Created_tmp_tables         | 3     |
| Handler_commit             | 1     |
| Handler_delete             | 0     |
| Handler_discover           | 0     |
| Handler_prepare            | 0     |
| Handler_read_first         | 1     |
| Handler_read_key           | 7483  |
| Handler_read_next          | 6462  |
| Handler_read_prev          | 0     |
| Handler_read_rnd           | 5462  |
| Handler_read_rnd_next      | 6478  |
| Handler_rollback           | 0     |
| Handler_savepoint          | 0     |
| Handler_savepoint_rollback | 0     |
| Handler_update             | 0     |
| Handler_write              | 6459  |
+----------------------------+-------+
```

Похоже, что в запросе использовались три временные таблицы — две из них на диске — и было много неиндексированных чтений (Handler_read_rnd_next). Если бы мы ничего не знали о представлении, к которому только что обращались, то могли бы предположить, что запрос сделал объединение без индекса, возможно, из-за подзапроса, который создал временные таблицы, а затем использовал их с правой стороны в соединении. Временные таблицы, созданные для хранения результатов подзапросов, не имеют индексов, поэтому эта версия кажется правдоподобной.

Используя эту методику, имейте в виду, что команда SHOW STATUS создает временную таблицу и обращается к ней с помощью обработчика операций, поэтому на полученные результаты в действительности влияет и SHOW STATUS. Это зависит от версий сервера. Используя информацию о выполнении запроса, полученную от команды SHOW PROFILES, мы можем предположить, что количество временных таблиц завышено на 2.

Стоит отметить, что большую часть той же информации, по-видимому, можно получить, просмотрев план EXPLAIN для этого запроса. Но EXPLAIN — это оценка того, что сервер планирует делать, а просмотр счетчиков статуса — это измерение того, что он на самом деле сделал. EXPLAIN не скажет вам, например, была ли временная таблица создана на диске, что медленнее, чем в памяти. Больше информации о команде EXPLAIN содержится в приложении Г.

Использование журнала медленных запросов

Что расширенный в Percona Server журнал медленных запросов расскажет об этом запросе? Вот что было зафиксировано при выполнении запроса, продемонстрированного в разделе о SHOW PROFILE:

```
# Time: 110905 17:03:18
# User@Host: root[root] @ localhost [127.0.0.1]
# Thread_id: 7  Schema: sakila  Last_errno: 0  Killed: 0
# Query_time: 0.166872  Lock_time: 0.000552  Rows_sent: 997  Rows_examined: 24861
  Rows_affected: 0  Rows_read: 997
# Bytes_sent: 216528  Tmp_tables: 3  Tmp_disk_tables: 2  Tmp_table_sizes: 11627188
# InnoDB_trx_id: 191E
# QC_Hit: No  Full_scan: Yes  Full_join: No  Tmp_table: Yes  Tmp_table_on_disk: Yes
# Filesort: Yes  Filesort_on_disk: No  Merge_passes: 0
#   InnoDB_IO_r_ops: 0  InnoDB_IO_r_bytes: 0  InnoDB_IO_r_wait: 0.000000
#   InnoDB_rec_lock_wait: 0.000000  InnoDB_queue_wait: 0.000000
#   InnoDB_pages_distinct: 20
# PROFILE_VALUES ... Copying to tmp table: 0.090623... [omitted]
SET timestamp=1315256598;
SELECT * FROM sakila.nicer_but_slower_film_list;
```

Похоже, что запрос действительно создал три временные таблицы, которые были скрыты от представления в SHOW PROFILE (возможно, из-за особенностей способа выполнения запроса сервером). Две временные таблицы находились на диске. Здесь мы сократили выведенную информацию для улучшения удобочитаемости. В конце концов, данные, полученные при выполнении команды SHOW PROFILE по этому запросу, записываются в журнал, поэтому вы можете журналировать в Percona Server даже такой уровень детализации.

Согласитесь, эта весьма подробная запись в журнале медленных запросов содержит практически все, что вы можете видеть в SHOW PROFILE и SHOW STATUS, и еще кое-что. Это делает журнал очень полезным для поиска более подробной информации при нахождении плохого запроса с помощью утилиты pt-query-digest. Когда вы просмотрите отчет от pt-query-digest, увидите такую строку заголовка:

```
# Query 1: 0 QPS, 0x concurrency, ID 0xEE758C5E0D7EADEE at byte 3214 _____
```

Вы можете использовать байтовое смещение для фокусировки на нужном разделе журнала следующим образом:

```
tail -c +3214 /path/to/query.log | head -n100
```

Вуаля! Можно рассмотреть все подробности. Кстати, pt-query-digest понимает все добавленные пары «имя — значение» формата медленного журнала запросов Percona Server и автоматически выводит намного более подробный отчет.

Использование Performance Schema

На момент написания этой книги таблицы Performance Schema, представленные в MySQL 5.5, не поддерживают профилирование на уровне запросов. Performance Schema появилась не так давно. Однако она быстро развивается, приобретая дополнительную функциональность в каждом следующем релизе. Но даже первоначальная функциональность MySQL 5.5 позволяет получать любопытную информацию. Например, следующий запрос покажет основные причины ожидания в системе:

```
mysql> SELECT event_name, count_star, sum_timer_wait
    -> FROM events_waits_summary_global_by_event_name
    -> ORDER BY sum_timer_wait DESC LIMIT 5;
+---------------------------------------+------------+------------------+
| event_name                            | count_star | sum_timer_wait   |
+---------------------------------------+------------+------------------+
| innodb_log_file                       | 205438     | 2552133070220355 |
| Query_cache::COND_cache_status_changed | 8405302    | 2259497326493034 |
| Query_cache::structure_guard_mutex    | 55769435   |  361568224932147 |
| innodb_data_file                      | 62423      |  347302500600411 |
| dict_table_stats                      | 15330162   |   53005067680923 |
+---------------------------------------+------------+------------------+
```

Сейчас существует несколько моментов, ограничивающих использование Performance Schema в качестве инструмента профилирования общего назначения. Во-первых, она не обеспечивает достаточный уровень детализации выполнения запросов и затрат времени, который можно получить благодаря существующим инструментам. Во-вторых, она довольно долго не использовалась и в данный момент ее применение приводит к бóльшим издержкам, чем применение привычного для многих инструмента профилирования. (Есть основания полагать, что это будет исправлено в ближайшее время.)

Наконец, иногда она слишком сложна и низкоуровнева для использования большинством пользователей. Функции, реализованные к настоящему моменту, в основном нацелены на то, что нужно измерить при изменении исходного кода MySQL для улучшения производительности сервера. Сюда относятся такие элементы, как ожидания и мьютексы. Некоторые из функций MySQL 5.5 полезны для опытных пользователей, а не для разработчиков серверов. Однако пользователи все еще нуждаются в разработке удобных инструментов интерфейса. В настоящее время для написания сложных запросов к разнообразным таблицам метаданных с большим количеством столбцов требуется настоящее мастерство. Это довольно сложный для использования и понимания набор инструментов.

Будет здорово, когда Performance Schema в более поздних версиях MySQL получит больше функциональности. И очень приятно, что Oracle реализует ее как таблицы, доступные через SQL, тем самым пользователи могут получать данные любым удобным для них способом. Однако пока она еще не способна заменить журнал медленных запросов или другие инструменты, помогающие сразу увидеть варианты улучшения производительности сервера и выполнения запросов.

Использование профиля для оптимизации

Итак, у вас есть профиль сервера или запроса — что с ним делать? Хороший профиль обычно делает проблему очевидной, но *решения* может и не быть (хотя чаще всего есть). На этом этапе, особенно при оптимизации запросов, вам нужно полагаться на знания о сервере и о том, как он выполняет запросы. Профиль или те данные, которые вы можете собрать, указывают направление движения и дают основания для применения ваших знаний и нахождения результатов с помощью дополнительных инструментов, таких как `EXPLAIN`. Подробнее эта тема будет рассмотрена в следующих главах, но сейчас у вас есть правильная отправная точка.

В общем, хотя нахождение источника проблемы с помощью профиля со всеми метриками не должно представлять труда, на деле невозможно выполнить измерения абсолютно точно, поскольку оцениваемые системы не поддерживают этой возможности. Ранее, рассматривая пример, мы подозревали, что на временные таблицы и неиндексированные чтения затрачивается бóльшая часть времени отклика, однако не можем этого доказать. Иногда проблемы трудно решить, потому что, возможно, не измерено все, что нужно, либо измерения сделаны в неверном направлении. Например, вы можете определять активность всего сервера вместо изучения того фрагмента, который пытаетесь оптимизировать, или анализировать измерения, проведенные с момента времени до начала выполнения запроса, а не тогда, когда он был запущен.

Существует еще одна возможность. Предположим, вы анализируете журнал медленных запросов и находите простой запрос, на несколько запусков которого затрачено неоправданно много времени, хотя он быстро запускался в тысячах других случаев. Вы снова запускаете запрос, и он выполняется молниеносно, как и должно быть. Применяете `EXPLAIN` и обнаруживаете, что он правильно использует индекс. Вы пытаетесь использовать похожие запросы с разными значениями в разделе `WHERE`, чтобы убедиться, что запрос не обращается к кэшу, и они тоже выполняются быстро. Кажется, что с этим запросом все нормально. Что дальше?

Если у вас есть только стандартный журнал медленных запросов MySQL без плана выполнения или подробной информации о времени, вы знаете только, что запрос плохо работал, когда был журналирован, и не можете понять, почему это произошло. Возможно, что-то еще потребляло ресурсы в системе, например резервное копирование или какая-то блокировка или параллелизм тормозили ход запроса. Периодически возникающие проблемы — это особый случай, который мы рассмотрим в следующем разделе.

Диагностика редко возникающих проблем

Редко возникающие проблемы, такие как случайный серверный стопор или медленное выполнение запросов, могут портить диагностику. Самые вопиющие потери времени, которые мы встречали, были результатом фантомных проблем, возникающих только тогда, когда за системой никто не следил, при этом воспроизвести их не удавалось. Нам доводилось видеть, как люди тратят целые месяцы на борьбу

с такими проблемами. В ходе этого некоторые специалисты устраняли неполадки методом проб и ошибок и иногда значительно ухудшали ситуацию, пытаясь случайным образом изменить настройки сервера и надеясь наткнуться на решение проблемы.

Старайтесь, насколько это возможно, избегать метода проб и ошибок. Такое устранение неполадок рискованно, поскольку результаты могут быть отрицательными и, как следствие, весь процесс окажется бесполезным и неэффективным. Если вы не можете понять, в чем проблема, возможно, вы неправильно выполняете измерения, делаете это в неправильном месте или не знаете, какие инструменты надо использовать. (Или последние могут отсутствовать — мы разработали ряд инструментов, предназначенных специально для увеличения прозрачности различных компонентов системы, от операционной системы до MySQL в целом.)

Чтобы проиллюстрировать необходимость отказаться от метода проб и ошибок, рассмотрим несколько практических примеров решения некоторых редко возникающих проблем с производительностью баз данных.

- Приложение выполняло утилиту curl для получения котировок обменных курсов от в разы более медленного внешнего сервиса.
- Важные записи кэша исчезли из memcached, в результате чего приложение переполняло MySQL запросами на восстановление кэшированных элементов.
- DNS-запросы синхронизировались случайным образом.
- Кэш запросов периодически замораживал MySQL из-за конкуренции мьютексов или неэффективных внутренних алгоритмов удаления кэшированных запросов.
- Ограничения масштабируемости InnoDB приводили к слишком длительной оптимизации плана запроса, если объем конкурентного доступа превышал некоторое пороговое значение.

Как видите, одни из этих проблем связаны с базой данных, а другие — нет. Только начав отладку с места, где наблюдается неправильное поведение системы, выяснив, какие ресурсы здесь используются, и выполнив максимально полные измерения, можно избежать поиска проблем там, где их нет.

Здесь мы закончим читать вам лекцию и начнем рассматривать подход и инструменты, которые обычно используем для решения редко возникающих проблем.

Проблемы одиночного запроса или всего сервера?

У вас есть какие-либо доказательства существования проблемы? Если да, попробуйте определить, связана она с изолированным запросом или со всем сервером. Это важно для понимания того, куда двигаться в ходе ее решения. Если на сервере все работает плохо, а затем нормализуется, тогда медленно работающий запрос вряд ли является источником проблемы. Большинство медленных запросов, скорее всего, сами стали жертвами какой-то другой проблемы. В то же время, если сервер в целом работает хорошо, а один запрос по какой-то причине выполняется медленно, следует внимательнее его изучить.

Проблемы с сервером возникают довольно часто. По мере появления в последние годы более мощного оборудования, когда 16-ядерные и более крупные серверы

становятся нормой, ограничения масштабируемости MySQL на SMP-системы стали более заметными. Истоки большинства этих проблем кроются в более старых версиях, которые, к сожалению, все еще широко используются. У MySQL еще есть проблемы с масштабируемостью даже в более новых версиях, но они гораздо менее серьезны и намного реже встречаются, поскольку являются пограничными случаями. Это и хорошая, и плохая новость: хорошая, потому что вероятность столкнуться с ними невелика, и плохая, так как для их диагностики требуется больше знаний о внутренних функциях MySQL. Это также означает, что множество проблем можно решить, просто обновив MySQL[1].

Как определить, является ли проблема общей для сервера или ограничена единственным запросом? Существуют три простых метода для выяснения этого. Мы рассмотрим их далее.

Команда SHOW GLOBAL STATUS

Суть методики заключается в очень частой фиксации выборки `SHOW GLOBAL STATUS`, например один раз в секунду и при появлении проблемы, поиске пиков или провалов в показаниях счетчиков, таких как `Threads_running`, `Threads_connected`, `Questions` и `Queries`. Это простой, не затрагивающий сервера метод, который доступен любому пользователю (никаких специальных привилегий не требуется), а поэтому — отличный способ больше узнать о природе проблемы, не затрачивая много времени. Приведем пример команды и вывода:

```
$ mysqladmin ext -i1 | awk '
    /Queries/{q=$4-qp;qp=$4}
    /Threads_connected/{tc=$4}
    /Threads_running/{printf "%5d %5d %5d\n", q, tc, $4}'
2147483647   136     7
  798   136     7
  767   134     9
  828   134     7
  683   134     7
  784   135     7
  614   134     7
  108   134    24
  187   134    31
  179   134    28
 1179   134     7
 1151   134     7
 1240   135     7
 1000   135     7
```

Команда каждую секунду фиксирует выборку `SHOW GLOBAL STATUS` и передает данные в скрипт awk, который выводит количество запросов в секунду, `Threads_connected` и `Threads_running` (количество запросов, выполняемых в данный момент). Эта троица, как правило, очень чувствительна к остановкам сервера. Обычно в зависимости от характера проблемы и способа подключения приложения к MySQL число запросов в секунду уменьшается, а по крайней мере один из двух

[1] Но не делайте этого, не имея весомых оснований полагать, что это решит проблему.

других показателей демонстрирует всплеск. В данном случае приложение, по-видимому, использует пул соединений, поэтому нет всплесков связанных потоков, но есть явная «кочка» в количестве выполняемых в этот момент запросов, при этом число запросов в секунду падает до доли от нормального уровня.

Как можно объяснить такое поведение? Рискованно строить догадки, но на практике мы встречались с двумя распространенными случаями. Один из них является своего рода внутренним узким местом на сервере: новые запросы начинают выполняться, но накапливаются у какой-то блокировки, которую ждут старые запросы. Этот тип блокировки обычно оказывает давление на серверы приложений и приводит к появлению там очередей. Другим распространенным случаем, который мы видели, является всплеск количества тяжелых запросов, например таких, которые могут появляться при неудачно настроенном устаревании memcached.

При скорости одна строка в секунду вы можете легко запустить процесс на несколько часов или дней и построить быстрый график, на котором можно будет увидеть наличие отклонений. Если проблема действительно периодическая, вы можете запустить систему на необходимое ей для работы время, а затем, когда заметите проблему, обратиться к выведенной информации. В большинстве случаев этот вывод ясно ее покажет.

Команда SHOW PROCESSLIST

С помощью этого метода вы фиксируете выборки `SHOW PROCESSLIST` и ищете множество потоков, которые находятся в необычных состояниях или имеют другие необычные характеристики. Например, довольно редко запросы остаются в состоянии «статистика» в течение длительного времени, поскольку это фаза оптимизации запросов, на которой сервер определяет лучший порядок соединения — обычно очень быстро.

Аналогично редко случается, что множество потоков отмечают пользователя как неаутентифицируемого. Дело в том, что это состояние характерно для установления связи при соединении, когда клиент определяет, какой пользователь пытается войти в систему.

Вертикальный вывод с помощью терминатора `\G` очень полезен для работы с `SHOW PROCESS LIST`, поскольку он помещает каждый столбец каждой строки вывода в собственную строку, что упрощает выполнение небольшого `sort|uniq|sort`-«заклинания», помогающего увидеть количество уникальных значений в любом желаемом столбце:

```
$ mysql -e 'SHOW PROCESSLIST\G' | grep State: | sort | uniq -c | sort -rn
    744   State:
     67   State: Sending data
     36   State: freeing items
      8   State: NULL
      6   State: end
      4   State: Updating
```

```
4  State: cleaning up
2  State: update
1  State: Sorting result
1  State: logging slow query
```

Если вы хотите изучить другой столбец, просто измените паттерн grep. Столбец `State` подходит для многих случаев. В данном примере мы видим огромное число потоков в состояниях, соответствующих окончанию запроса: «освобождение элементов», «конец», «очистка» и «журналирование медленных запросов». Фактически во многих примерах на сервере, работа которого отражена в выведенной информации, наблюдалась такая же или подобная ситуация. Самым характерным и надежным индикатором проблемы было большое количество запросов в состоянии освобождения элементов.

Вам не надо использовать методики командной строки для поиска подобных проблем. Можете сделать запрос к таблице `PROCESSLIST` в `INFORMATION_SCHEMA`, если сервер довольно новый, или использовать утилиту innotop с большой частотой обновления и наблюдать на экране за необычным скоплением запросов. Только что рассмотренный пример был получен на сервере с проблемами внутреннего параллелизма и очистки InnoDB, но такая же ситуация может сложиться и в других обстоятельствах. Классическим примером было бы множество запросов в состоянии «заблокировано». Это неизменный отличительный признак MyISAM с ее блокировкой на уровне таблиц, быстро приводящей к захламлению сервера в случае большого количества записей в таблицах.

Использование журналирования запросов

Чтобы найти проблемы в журнале запросов, включите журнал медленных запросов, установите значение `long_query_time`, глобально равное `0`, и убедитесь, что все соединения видят новый параметр. Возможно, вам придется перезапустить соединения, чтобы они могли увидеть новое глобальное значение, или использовать функцию из пакета Percona Server, чтобы заставить изменения вступить в силу мгновенно, без прерывания существующих соединений.

Если по какой-либо причине вы не можете включить журнал медленных запросов для фиксации всех запросов, используйте для его эмуляции утилиты tcpdump и pt-query-digest. Ищите в журнале интервалы, на которых пропускная способность внезапно падает. Запросы отправляются в журнал медленных запросов по мере завершения транзакции, поэтому захламление обычно приводит к внезапному падению количества завершенных транзакций, которое продолжается до тех пор, пока виновник не завершит работу и не освободит ресурс, блокирующий другие запросы. Затем будут завершены другие запросы. Что полезно в таком характерном поведении, так это то, что вы можете обвинить в захламлении первый запрос, который завершается после снижения пропускной способности. (Иногда это не самый первый запрос — часть запросов могут продолжать работать, пока другие заблокированы, поэтому такой подход не будет надежным в любой ситуации.)

И вновь вам могут пригодиться хорошие инструменты. Вы не можете самолично просматривать сотни гигабайт запросов. Вот однострочный пример, который основан на паттерне MySQL для записи текущего времени в журнал через 1 секунду:

```
$ awk '/^# Time:/{print $3, $4, c;c=0}/^# User/{c++}' slow-query.log
080913 21:52:17 51
080913 21:52:18 29
080913 21:52:19 34
080913 21:52:20 33
080913 21:52:21 38
080913 21:52:22 15
080913 21:52:23 47
080913 21:52:24 96
080913 21:52:25 6
080913 21:52:26 66
080913 21:52:27 37
080913 21:52:28 59
```

Как видите, здесь наблюдалось снижение пропускной способности, но, что интересно, ей предшествовал всплеск числа завершенных запросов. Не заглянув в журнал, трудно сказать, что произошло, но, возможно, этот всплеск был связан с последующим падением. В любом случае ясно, что на сервере произошло что-то странное, и исследование журнала в области соответствующих временных меток может оказаться очень плодотворным. (Просмотрев этот журнал, мы обнаружили, что всплеск произошел из-за отключения соединений. Возможно, сервер приложений был перезапущен. Не все, что происходит, оказывается проблемой MySQL.)

Осмысление результатов

Визуализация данных — незаменимая операция. В этой книге мы приводим лишь небольшие примеры, но на практике многие из рассмотренных методик могут вызвать появление тысяч строк вывода. Познакомьтесь с gnuplot, или R, или другими графическими инструментами по своему выбору. Вы можете использовать их для моментального создания графика — намного быстрее, чем в электронной таблице, — и мгновенно приблизить на нем отклонения, что намного труднее сделать, прокручивая строки, даже если вы думаете, что хороши в просматривании «матрицы»[1].

Для начала рекомендуем попробовать первые два подхода: `SHOW STATUS` и `SHOW PROCESSLIST`, потому что они малозатратны и могут быть выполнены в интерактивном режиме всего лишь со сценарием оболочки или повторным выполнением запросов. Анализировать журнал медленно работающих запросов гораздо труднее — при этом часто можно наблюдать какие-то странные закономерности, при более внимательном изучении пропадающие. Мы обнаружили, что легко представить себе закономерности там, где их нет.

Если вы обнаружите отклонение, что это будет означать? Обычно то, что запросы где-то в очереди или есть поток или всплеск определенного типа запросов. Дальнейшая задача состоит в том, чтобы обнаружить его причину.

[1] Мы еще не видели женщину в красном платье, но, как увидим, сразу вам сообщим.

Фиксация данных диагностики

При возникновении редко появляющейся проблемы важно измерить *все*, что возможно, и желательно во время ее проявления. Если вы сделаете это правильно, то соберете огромное количество данных диагностики. Сведения, которые вы не собираете, часто оказываются именно тем, что вам действительно нужно для правильного определения проблемы.

Для начала вам понадобятся две вещи.

- Надежный и действующий в реальном времени триггер — способ узнать, когда возникает проблема.
- Инструмент для сбора данных диагностики.

Диагностический триггер

Триггер очень важен для правильной работы. Это основа для фиксации данных при возникновении проблемы. Есть две основные вещи, мешающие триггеру работать правильно: ложноположительные и ложноотрицательные результаты. Если у вас есть ложноположительный результат, вы станете собирать данные диагностики, хотя ничего плохого не случилось. Тем самым будете напрасно тратить время и расстраиваться. Из-за ложноотрицательных результатов вы упустите возможность найти проблему, потратите еще больше времени и серьезнее огорчитесь. Потратьте немного времени и убедитесь, что ваш триггер однозначно указывает на появление проблемы. Это будет не напрасным.

Что такое хороший критерий для тригтера? Как показано в наших примерах, `Threads_running`, как правило, очень чувствителен к проблемам, но довольно стабилен, когда все хорошо работает. Еще одним отличным показателем является всплеск необычных состояний потоков в `SHOW PROCESSLIST`. Помимо них, существует еще много способов наблюдать за этой проблемой, в том числе специальный вывод в `SHOW INNODB STATUS`, всплеск средней загрузки сервера и т. д. Ключевой момент состоит в получении чего-то, что можно сравнить с определенным порогом. Обычно это означает подсчет. Подойдет подсчет работающих потоков, подсчет потоков в состоянии освобождения элементов и т. п. При просмотре состояний потоков вам поможет параметр `-c` для `grep`:

```
$ mysql -e 'SHOW PROCESSLIST\G' | grep -c "State: freeing items"
36
```

Выберите такое пороговое значение, которое было бы достаточно высоким, чтобы его нельзя было достигнуть во время нормальной работы, но и не настолько высоким, чтобы нельзя было зафиксировать проблему, когда она появится. Остерегайтесь также устанавливать слишком высокое пороговое значение, поскольку в противном случае вы не обнаружите проблему сразу после ее появления. Обостряющиеся проблемы, как правило, вызывают каскад других проблем, и если вы зафиксируете диагностическую информацию только после того, как все полетело в тартарары, вам, вероятно, будет труднее выявить исходную причину. Крайне желательно получить данные, когда вода просто стекает в раковину — до того, как громкий смыв оглушит

вас Например, всплески в `Threads_connected` могут быть невероятно высокими — мы видели, как они увеличивались с 100 до 5000 и даже больше в течение пары минут. Вы могли бы использовать 4999 в качестве порогового значения, но зачем ждать, когда все станет настолько плохо? Если приложение, будучи работающим, не открывает более 150 соединений, начните сбор с 200 или 300.

Возвращаясь к примеру с `Threads_running`, отметим, что, похоже, нормальный уровень параллелизма — менее 10. Но 10 не будет хорошим пороговым значением — в этом случае существует большая вероятность ложноположительных результатов, и 15 недостаточно далеко, чтобы определенно быть вне нормального диапазона поведения. При 15 может произойти мини-захламление, но весьма вероятно, что красная линия не будет пересечена и проблема может рассосаться сама собой, прежде чем ситуация станет настолько критической, чтобы ее можно было четко диагностировать. В данном примере мы бы предложили установить в качестве порогового значения 20.

По-видимому, вы хотели бы зафиксировать проблему, как только она появится, но только после небольшого ожидания, чтобы убедиться, что это не ложноположительный результат или кратковременный всплеск. Итак, наш последний триггер такой: следите за статусом переменных с периодичностью один раз в секунду и, если `Threads_running` будет превышать показатель 20 в течение более 5 секунд, начните сбор диагностических данных. (Кстати, пример показал, что проблема исчезает через 3 секунды. Мы слегка подогнали его, чтобы сделать небольшим. Трехсекундную проблему, как правило, трудно диагностировать. Однако большинство проблем, которые нам встречались, длятся немного дольше.)

Теперь следует настроить какой-то инструмент для просмотра сервера и реагирования на срабатывания триггера. Вы могли бы написать его самостоятельно, но мы уже сделали это. В пакете Percona Toolkit есть инструмент pt-stalk, специально предназначенный для этих целей. У него много приятных функций, необходимость которых мы ощутили на собственной шкуре. Например, он следит за объемом свободного места на диске, поэтому не заполнит диск собранными данными и не обрушит сервер. Вы же понимаете, что мы этого никогда не делали!

Инструмент pt-stalk очень прост в использовании. Вы можете установить параметры для просмотра, порогового значения, частоты проверок и т. п. У него намного больше замечательных возможностей, но для данного примера этого будет достаточно. Прежде чем применять его, прочтите руководство пользователя. Он опирается на другой инструмент для сбора данных, который мы обсудим далее.

Какие данные следует собирать

Теперь, когда вы определились с диагностическим триггером, можете использовать его для запуска процессов сбора данных. Но какие данные вы должны собирать? Ответ на этот вопрос уже звучал: все, что можете, но только за разумное время. Собирайте статистику операционной системы, данные об использовании процессора, диска и о свободном месте на нем, об использовании памяти, выборки вывода ps и все, что можете получить от MySQL, например выборки `SHOW STATUS`, `SHOW PROCESSLIST`

и `SHOW INNODB STATUS`. Вся эта информация, а возможно, и еще какая-нибудь, вам понадобится для диагностики проблемы.

Как вы помните, время выполнения делится на работу и ожидание. В общем случае есть два типа причин возникновения неизвестной проблемы. Сервер может выполнять много работы, потребляя много циклов процессора, или застрять в ожидании освобождения ресурсов. Необходимо применять два разных подхода, чтобы собрать диагностические данные для определения причин каждого из этих типов проблем. Вам нужен профиль, если система выполняет слишком много работы, и анализ ожиданий, если она слишком долго ждет. Но как узнать, на чем следует сосредоточиться, если проблема неизвестна? Никак, поэтому лучше собрать данные для обоих типов проблем.

Первичным инструментом профилирования, который мы используем для внутренних серверов на GNU/Linux (в отличие от запросов по всему серверу), является oprofile. Примеры его применения будут показаны чуть позже. Можно также профилировать системные вызовы сервера с помощью инструмента strace, но мы обнаружили, что это рискованно для систем в рабочем состоянии. Об этом тоже поговорим позже. Для фиксации запросов в профиль мы предпочитаем использовать утилиту tcpdump. В большинстве версий MySQL журнал медленных запросов тяжело быстро включать и выключать, однако вы можете хорошо имитировать этот процесс с помощью трафика TCP. Кроме того, трафик полезен для многих других видов анализа.

Для анализа ожиданий мы обычно используем трассировки стека GDB[1]. Потоки, застрявшие в одном месте внутри MySQL на протяжении длительного времени, как правило, имеют одну и ту же трассировку стека. Необходимо запустить gdb, прикрепить его к процессу mysqld и сбрасывать трассировки стека для всех потоков. Затем вы можете использовать короткие скрипты для объединения общих трассировок стека и с помощью волшебных `sort|uniq|sort` показать, какие из них имеют больше всего общих черт. Чуть позже мы объясним, как использовать для этого инструмент pt-pmp.

Вы также можете анализировать ожидания с моментальными снимками, сделанными командами `SHOW PROCESSLIST` и `SHOW INNODB STATUS`, наблюдая за потоками и состояниями транзакций. Ни один из этих подходов не является абсолютно надежным, но на практике их применяют довольно часто, поскольку они весьма полезны.

Кажется, что сбор всех этих данных требует большой работы! Вероятно, вы уже этого ждете, но мы создали инструмент и для этих целей. Он называется pt-collect и также является частью пакета Percona Toolkit. Предназначен для выполнения совместно с pt-stalk. Чтобы собрать большинство важных данных, его следует запустить как root. По умолчанию он будет собирать данные в течение 30 с, а затем

[1] Оговоримся, что использование GDB можно приравнять к хирургическому вмешательству. Стек мгновенно замораживает сервер, особенно если у вас много потоков (соединений), а иногда может даже обрушить его. Тем не менее преимущества иногда сводятся на нет из-за риска. Если сервер в любом случае становится непригодным для использования из-за стопора, нет ничего плохого в его двойном замораживании.

прекратит работу. Обычно этого бывает достаточно для диагностирования большинства проблем. В то же время это не так долго, чтобы получить ложнопозитивный результат.

Инструмент легко скачать, он не нуждается в какой-либо конфигурации — вся конфигурация переходит в pt-stalk. Убедитесь, что на вашем сервере установлены gdb и oprofile, и включите их в конфигурацию pt-stalk. Следует также убедиться, что mysqld содержит отладочные символы[1]. Когда возникнет условие срабатывания триггера, инструмент соберет довольно полный набор данных. В определенном каталоге он создаст файлы с временными метками. На момент написания книги инструмент ориентирован скорее на GNU/Linux и нуждается в настройке на другие операционные системы, тем не менее его стоит использовать.

Интерпретация данных

Если вы правильно настроили условие триггера и позволили pt-stalk работать достаточно долго для того, чтобы несколько раз обнаружить проблему, вы получите много данных для проверки. С чего следует начать? Во-первых, убедитесь, что проблема действительно возникла, поскольку, имея множество выборок для проверки, вы вряд ли захотите тратить время на ложнопозитивные результаты. Во-вторых, посмотрите, не бросается ли в глаза что-то очевидное.

Очень полезно фиксировать, как выглядит сервер, когда все работает хорошо, а не только в случае неприятностей. Это помогает определить, является конкретный образец или даже часть образца ненормальным или нет. Например, когда вы просматриваете состояния запросов в списке процессов, можете ответить на вопрос: «Нормально ли, что многие запросы сортируют свои результаты?»

Полезнее всего обращать внимание на поведение запросов и транзакций, а также внутреннее поведение сервера. Запрос и транзакция показывают, вызвана ли проблема тем, как используется сервер: плохо написанным SQL, плохим индексированием, плохим проектированием логической базы данных и т. д. Вы можете видеть, что пользователи делают с сервером, просматривая места, в которых появляются запросы и транзакции: журналированный трафик TCP, вывод команды `SHOW PROCESSLIST` и т. д. Внутреннее поведение сервера свидетельствует о том, есть ли на нем ошибки либо проблемы с производительностью или масштабируемостью. Вы можете обнаружить ошибки в разных местах, а также в выводах oprofile и gdb. Но для интерпретации таких результатов требуется опыт.

[1] Иногда символы отсутствуют из-за «оптимизации», которая на самом деле не является оптимизацией, — это только затрудняет диагностику проблем. Вы можете использовать инструмент nm, чтобы проверить, есть ли они у вас, и установить пакеты debuginfo для MySQL с целью получения символов.

Если вы не знаете, как интерпретировать полученную ошибку, то можете заархивировать каталог с собранными данными и отправить его на анализ в службу поддержки. Любой компетентный специалист по поддержке MySQL сможет истолковать данные и объяснить вам, что они значат. Кроме того, он будет только рад, что вы отправили для ознакомления такие подробные данные. Возможно, вы также захотите отправить вывод двух других инструментов пакета Percona Toolkit — pt-mysql-summary и pt-summary. Они показывают моментальные снимки статуса и конфигурации вашего экземпляра MySQL, операционной системы и оборудования соответственно.

Пакет Percona Toolkit включает в себя инструмент, который поможет вам быстро просмотреть множество выборок собранных данных. Он называется pt-sift, помогает перемещаться между выборками, показывает сводку каждой выборки и позволяет при необходимости погрузиться в отдельные биты данных. Этот инструмент может сэкономить много времени.

Ранее мы приводили примеры счетчиков статуса и состояний потоков. В заключение этой главы покажем несколько примеров вывода инструментов oprofile и gdb. Вот выданный oprofile отчет с сервера, на котором возникла проблема. Вы можете ее найти?

```
samples  %        image name   app name     symbol  name
893793   31.1273  /no-vmlinux  /no-vmlinux  (no symbols)
325733   11.3440  mysqld       mysqld       Query_cache::free_memory_
                                            block()
117732    4.1001  libc         libc         (no symbols)
102349    3.5644  mysqld       mysqld       my_hash_sort_bin
 76977    2.6808  mysqld       mysqld       MYSQLparse()
 71599    2.4935  libpthread   libpthread   pthread_mutex_trylock
 52203    1.8180  mysqld       mysqld       read_view_open_now
 46516    1.6200  mysqld       mysqld       Query_cache::invalidate_query_
                                            block_list()
 42153    1.4680  mysqld       mysqld       Query_cache::write_result_
                                            data()
 37359    1.3011  mysqld       mysqld       MYSQLlex()
 35917    1.2508  libpthread   libpthread   __pthread_mutex_unlock_usercnt
 34248    1.1927  mysqld       mysqld       __intel_new_memcpy
```

Если вы ответили «кэш запросов», вы правы. Кэш запросов этого сервера порождал слишком много работы и все тормозил. Замедление в 50 раз произошло мгновенно, без каких-либо других изменений в системе. Отключение кэша запросов нормализовало работу сервера. В этом примере интерпретировать внутреннюю среду сервера оказалось довольно просто.

Другим важным инструментом анализа узких мест является рассмотрение ожиданий с трассировкой стека с помощью gdb. Трассировка стека одного потока обычно выглядит следующим образом (мы немного отформатировали ее для печати):

```
Thread 992 (Thread 0x7f6ee0111910 (LWP 31510)):
#0  0x0000003be560b2f9 in pthread_cond_wait@@GLIBC_2.3.2 () from /libpthread.so.0
#1  0x00007f6ee14f0965 in os_event_wait_low () at os/os0sync.c:396
#2  0x00007f6ee1531507 in srv_conc_enter_innodb () at srv/srv0srv.c:1185
#3  0x00007f6ee14c906a in innodb_srv_conc_enter_innodb () at handler/ha_innodb.cc:609
#4  ha_innodb::index_read () at handler/ha_innodb.cc:5057
#5  0x00000000006538c5 in ?? ()
```

```
#6  0x0000000000658029 in sub_select() ()
#7  0x0000000000658e25 in ?? ()
#8  0x00000000006677c0 in JOIN::exec() ()
#9  0x000000000066944a in mysql_select() ()
#10 0x0000000000669ea4 in handle_select() ()
#11 0x00000000005ff89a in ?? ()
#12 0x0000000000601c5e in mysql_execute_command() ()
#13 0x000000000060701c in mysql_parse() ()
#14 0x000000000060829a in dispatch_command() ()
#15 0x0000000000608b8a in do_command(THD*) ()
#16 0x00000000005fbd1d in handle_one_connection ()
#17 0x0000003be560686a in start_thread () from /lib64/libpthread.so.0
#18 0x0000003be4ede3bd in clone () from /lib64/libc.so.6
#19 0x0000000000000000 in ?? ()
```

Стек читают снизу вверх, то есть в настоящее время выполняется поток внутри функции `pthread_cond_wait`, которая была вызвана из `os_event_wait_low`.

Считывая трассировку дальше, мы видим, что, судя по всему, этот поток пытался войти в ядро InnoDB (`srv_conc_enter_innodb`), но попал во внутреннюю очередь (`os_event_wait_low`), потому что в ядре уже было больше потоков, чем указано в переменной `innodb_thread_concurrency`. Однако реальное значение трассировки стека объединяет их. Это методика, которую Домас Митузас (Domas Mituzas), бывший инженер поддержки MySQL, сделал популярной благодаря инструменту «профилировщик для бедных». В настоящее время он работает в Facebook и вместе с коллегами разработал множество инструментов для сбора и анализа трассировок стека. Вы можете больше узнать о существующих инструментах на сайте http://www.poormansprofiler.org.

В пакете Percona Toolkit есть наша реализация «профилировщика для бедных» под названием pt-pmp. Это оболочка и программа awk, которая объединяет аналогичные трассировки стека и выполняет обычные команды `sort|uniq|sort`, показывая первыми наиболее часто встречающиеся. Далее показано, как выглядит полный набор трассировок стека. Мы будем использовать параметр `-l 5` для отсечения трассировки стека после пяти уровней так, чтобы не было большого количества трассировок с общими вершинами, но с разными основаниями, что могло бы помешать объединить их и увидеть, где на самом деле наблюдаются ожидания:

```
$ pt-pmp -l 5 stacktraces.txt
    507 pthread_cond_wait,one_thread_per_connection_end,
        handle_one_connection,start_thread,clone
    398 pthread_cond_wait,os_event_wait_low,srv_conc_enter_innodb,
        innodb_srv_conc_enter_innodb,ha_innodb::index_read
     83 pthread_cond_wait,os_event_wait_low,sync_array_wait_event,
        mutex_spin_wait,mutex_enter_func
     10 pthread_cond_wait,os_event_wait_low,os_aio_simulated_handle,
        fil_aio_wait,io_handler_thread
      7 pthread_cond_wait,os_event_wait_low,srv_conc_enter_innodb,
        innodb_srv_conc_enter_innodb,ha_innodb::general_fetch
      5 pthread_cond_wait,os_event_wait_low,sync_array_wait_event,
        rw_lock_s_lock_spin,rw_lock_s_lock_func
      1 sigwait,signal_hand,start_thread,clone,??
      1 select,os_thread_sleep,srv_lock_timeout_and_monitor_thread,
        start_thread,clone
```

```
1 select,os_thread_sleep,srv_error_monitor_thread,start_thread,clone
1 select,handle_connections_sockets,main
1 read,vio_read_buff,::??,my_net_read,cli_safe_read

1 pthread_cond_wait,os_event_wait_low,sync_array_wait_event,
  rw_lock_x_lock_low,rw_lock_x_lock_func
1 pthread_cond_wait,MYSQL_BIN_LOG::wait_for_update,mysql_binlog_send,
  dispatch_command,do_command
1 fsync,os_file_fsync,os_file_flush,fil_flush,log_write_up_to
```

Первая строка является характерной сигнатурой простаивающего потока в MySQL, поэтому можете ее проигнорировать. Наиболее интересна вторая строка: она показывает, что множество потоков ожидают входа в ядро InnoDB, но блокируются. Третья строка показывает множество потоков, ожидающих некоторого мьютекса, но какого именно, мы увидеть не можем, поскольку отсекли более глубокие уровни трассировки стека. Если важно узнать, какой это мьютекс, следует перезапустить инструмент с бóльшим значением параметра `-l`. В целом трассировки стека показывают, что множество потоков ждут своей очереди внутри InnoDB. Но почему так происходит? Это непонятно. Чтобы ответить на этот вопрос, нам, по-видимому, нужно поискать в другом месте.

Как видно из отчетов по трассировке стека и отчетов oprofile, такой вид анализа не всегда подходит для тех, кто не разбирается в исходном коде MySQL и InnoDB. В этом случае придется кого-то звать на помощь.

Теперь рассмотрим сервер, проблемы которого не отображаются ни в профиле, ни в анализе ожиданий. Их придется диагностировать другими методами.

Кейс по диагностике

Рассмотрим процесс диагностики редко возникающих проблем с производительностью на примере конкретной ситуации. В этом кейсе мы заберемся в неизвестную вам область, если вы не эксперт в MySQL, InnoDB и GNU/Linux. Однако основная идея этого кейса в другом. Попытайтесь найти метод в безумии: прочитайте этот раздел, ориентируясь на сделанные нами предположения и догадки. Рассмотрите подходы, основанные на рассуждениях и измерениях. Мы детально вникаем в конкретный случай лишь для иллюстрации общих положений.

Прежде чем приступить к решению проблемы, откликаясь на чью-то просьбу, стоит попробовать прояснить две вещи, желательно делая заметки, чтобы не забыть о чем-то важном.

1. В чем заключается проблема? Постарайтесь выяснить это. Удивительно легко начать охоту не за той проблемой. В данном случае клиент жаловался, что один раз в день или раз в два дня сервер отклоняет соединения, выдавая ошибку `max_connections`. Проблема появляется периодически и длится от нескольких секунд до нескольких минут.
2. Что было сделано в попытке это исправить? В данном случае клиент вообще не пытался решить проблему. Слушая о проблеме в изложении другого, очень

трудно понять, какова точная последовательность событий, в чем заключаются изменения, которые эти события повлекли за собой, и каков результат их воздействия. Это особенно верно, когда человек в отчаянии после пары бессонных ночей и до краев налитых кофеином дней. На сервере, с которым происходили неизвестные изменения с неизвестным эффектом, очень сложно устранять неполадки, особенно в цейтноте.

Обсудив эти моменты, приступим. Стоит не только попытаться понять, как ведет себя сервер, но и оценить его состояние, конфигурацию, программное обеспечение и оборудование. Мы сделали это с помощью инструментов pt-summary и pt-mysql-summary. На этом сервере были 16-ядерный процессор, 12 Гбайт ОЗУ и всего 900 Мбайт данных, все в InnoDB, на твердотельном диске. На сервере запущена GNU/Linux с MySQL 5.1.37 и плагином InnoDB версии 1.0.4. Ранее мы работали с этим клиентом, решая другие неожиданные проблемы с производительностью, и знали систему. В прошлом с базой данных никогда не было проблем — проблемы всегда возникали в приложении. Мы посмотрели на сервер и с первого взгляда не обнаружили ничего очевидного. Запросы не были идеальными, но в большинстве случаев все еще выполнялись быстрее 10 миллисекунд. Таким образом мы убедились, что в нормальных условиях сервер работал хорошо. (Это необходимо сделать — многие появляющиеся время от времени проблемы оказываются симптомами хронических проблем, таких как сбой жестких дисков в RAID-массивах.)

Этот кейс может оказаться немного утомительным. Мы будем валять дурака, чтобы показать все диагностические данные, подробно объяснить все, что обнаружили, и проследить разные направления мысли. На самом деле не используем столь удручающе медленный подход к решению проблемы и не пытаемся сказать, что вам это нужно.

Мы установили диагностический пакет и в качестве триггера выбрали `Threads_connected`, значение которого обычно составляет менее 15, но во время этих проблем увеличивалось до нескольких сотен. Покажем, пока без комментариев, выборку данных, которые мы получили. Посмотрите, сможете ли вы выбрать важное для дальнейшего анализа.

- Активность работы варьировалась от 1000 до 10 000 запросов в секунду, причем многие из них были мусорными командами, например пингование сервера для проверки того, работает ли он. Среди остальных преобладали команды `SELECT` — от 300 до 2000 в секунду. Также было немного команд `UPDATE` (около пяти в секунду).
- В команде `SHOW PROCESSLIST` было два основных типа запросов, различающихся только параметрами в разделе `WHERE`. Приведем обобщенные состояния запросов:

```
$ grep State: processlist.txt | sort | uniq -c | sort -rn
    161   State: Copying to tmp table
    156   State: Sorting result
    136   State: statistics
```

```
50   State: Sending data
24   State: NULL
13   State:
 7   State: freeing items
 7   State: cleaning up
 1   State: storing result in query cache
 1   State: end
```

- Большинство запросов сканировали индексы или диапазоны — полнотабличных и сканирований по кросс-соединениям не было.
- Во время работы происходило от 20 до 100 сортировок в секунду, при этом сортировалось от 1000 до 12 000 рядов в секунду.
- Во время работы создавалось от 12 до 90 временных таблиц в секунду, в том числе от 3 до 5 — на диске.
- Не было проблем с блокировкой таблиц или кэшем запросов.
- Выполнив команду `SHOW INNODB STATUS`, мы заметили, что состоянием главного потока является сброс страниц буферного пула, но имелось всего несколько десятков «грязных» страниц для сброса (`Innodb_buffer_pool_pages_dirty`), практически не было изменений в `Innodb_buffer_pool_pages_flushed`, а разница между порядковым номером последней записи в журнале и последней контрольной точкой оказалась небольшой. Буферный пул InnoDB даже не приблизился к заполнению — он был намного больше размера данных. Большинство потоков ожидали в очереди InnoDB: «12 запросов внутри InnoDB, 495 запросов в очереди».
- Мы зафиксировали вывод утилиты iostat в течение 30 секунд, одну выборку в секунду. Он показал, что на дисках практически не было чтения, а записей — выше крыши. При этом среднее время ожидания ввода/вывода и длина очереди были чрезвычайно высокими. Приведем фрагмент вывода, немного упростив его, чтобы он поместился на странице без переносов:

```
r/s     w/s rsec/s    wsec/s avgqu-sz  await svctm  %util
1.00 500.00   8.00  86216.00     5.05  11.95  0.59  29.40
0.00 451.00   0.00 206248.00   123.25 238.00  1.90  85.90
0.00 565.00   0.00 269792.00   143.80 245.43  1.77 100.00
0.00 649.00   0.00 309248.00   143.01 231.30  1.54 100.10
0.00 589.00   0.00 281784.00   142.58 232.15  1.70 100.00
0.00 384.00   0.00 162008.00    71.80 238.39  1.73  66.60
0.00  14.00   0.00    400.00     0.01   0.93  0.36   0.50
0.00  13.00   0.00    248.00     0.01   0.92  0.23   0.30
0.00  13.00   0.00    408.00     0.01   0.92  0.23   0.30
```

- Вывод утилиты vmstat подтвердил то, что мы видели в iostat, и показал, что процессоры в основном бездействуют, за исключением некоторого ожидания ввода/вывода во время всплеска записи (до 9 % ожидания).

Мозги еще не закипели? Это быстро происходит, когда вы глубоко погружаетесь в систему и у вас нет никаких предубеждений (или вы пытаетесь игнорировать их), поэтому в итоге смотрите на *все*. Большинство того, что вы увидите, либо совершенно нормально, либо показывает последствия проблемы, но ничего не говорит о ее источнике. Хотя в данном случае у нас есть некоторые догадки о причине проблемы,

продолжим исследование, обратившись к отчету oprofile. Но сейчас мы не только вывалим на вас новые данные, но и добавим к ним комментарии и интерпретацию:

```
samples  %        image name       app name         symbol name
473653   63.5323  no-vmlinux       no-vmlinux       /no-vmlinux
 95164   12.7646  mysqld           mysqld           /usr/libexec/mysqld
 53107    7.1234  libc-2.10.1.so   libc-2.10.1.so   memcpy
 13698    1.8373  ha_innodb.so     ha_innodb.so     build_template()
 13059    1.7516  ha_innodb.so     ha_innodb.so     btr_search_guess_on_hash
 11724    1.5726  ha_innodb.so     ha_innodb.so     row_sel_store_mysql_rec
  8872    1.1900  ha_innodb.so     ha_innodb.so     rec_init_offsets_comp_ordinary
  7577    1.0163  ha_innodb.so     ha_innodb.so     row_search_for_mysql
  6030    0.8088  ha_innodb.so     ha_innodb.so     rec_get_offsets_func
  5268    0.7066  ha_innodb.so     ha_innodb.so     cmp_dtuple_rec_with_match
```

Далеко не очевидно, что скрывается за большинством этих наборов символов, и большую часть времени они болтаются вместе в ядре[1] и в наборе символов mysqld, который ни о чем нам не говорит[2]. Не отвлекайтесь на все эти обозначения `ha_innodb.so`. Посмотрите на процент времени, которое они тратят: независимо от того, что они делают, это занимает так мало времени, что можно быть уверенными: не это источник проблемы. Перед нами пример проблемы, которую мы не решим с помощью анализа профиля. Мы смотрим на неверные данные. Когда вы видите что-то похожее на предыдущую выборку, продолжайте исследование, возьмите другие данные — возможно, там будет более очевидное указание на причину проблемы.

А сейчас, если вам интересен анализ ожиданий с помощью трассировки стека утилитой gdb, обратитесь к концу предыдущего раздела. Приведенная там выборка взята из системы, которую мы сейчас диагностируем. Если помните, основная часть трассировок стека просто ждала входа в ядро InnoDB, что соответствует 12 запросам внутри InnoDB, 495 запросам в очереди, полученным в результатах, выведенных командой `SHOW INNODB STATUS`.

Видите ли вы что-нибудь, явно указывающее на конкретную проблему? Мы не увидели — обнаружили лишь возможные симптомы множества проблем и по меньшей мере две их потенциальные причины. При этом мы опирались на интуицию и свой опыт. А еще заметили нечто не имеющее смысла. Если вы снова посмотрите на вывод утилиты iostat, то в столбце `wsec/s` увидите, что в течение примерно 6 секунд сервер записывает на диски сотни мегабайт данных в секунду. Каждый сектор состоит из 512 байт, поэтому выборка свидетельствует о том, что за секунду порой записывается до 150 Мбайт. Но вся база данных содержит всего 900 Мбайт, а рабочая нагрузка — это в основном запросы `SELECT`. Как такое могло произойти?

Исследуя систему, постарайтесь задать себе вопрос, есть ли что-то, что просто не сходится, и разберите это подробнее. Старайтесь доводить ход мысли до конца

[1] Теоретически нам нужны эти наборы символов ядра, чтобы понять, что происходит внутри него. На практике это трудно определить, а мы знаем, глядя на vmstat, что системный процессор использовался незначительно, поэтому вряд ли обнаружим какую-то деятельность, отличную от сна.

[2] Кажется, что это MySQL, написанная с ошибкой.

и не слишком часто отвлекаться на смежные темы, иначе просто можете забыть о многообещающих идеях. Делайте небольшие заметки и просматривайте их, чтобы убедиться, что вы расставили все точки над «i»[1].

На этом этапе мы могли бы перескочить прямо к выводу, но это было бы неверно. Из состояния основного потока видно, что InnoDB пытается сбросить «грязные» страницы, которые обычно не отображаются в выводе состояния, если только сброс не откладывается. Мы знаем, что эта версия InnoDB подвержена проблеме яростного сброса, известной также как *стопор контрольной точки*. Это то, что происходит, когда InnoDB не выполняет сбросы равномерно в течение некоторого времени, а неожиданно решает вынудить контрольную точку сделать это (очистить много данных). Это может вызвать серьезную блокировку внутри InnoDB, заставив все процессы встать в очередь и ждать входа в ядро, и, таким образом, образовать пробку на уровнях выше InnoDB на сервере. В главе 2 мы продемонстрировали пример периодических падений производительности, которые могут произойти в случае яростного сброса. Большинство симптомов, наблюдаемых на этом сервере, похожи на то, что происходит при принудительной установке контрольной точки, но в данном случае проблема в другом. Это можно доказать разными способами, самый, вероятно, простой из которых — посмотреть на счетчики команды `SHOW STATUS` и отследить изменения в счетчике `Innodb_buffer_pool_pages_flushed`, который, как мы сказали ранее, не увеличился. Кроме того, мы отметили, что в пуле буферов не так много черновых данных, чтобы сбрасывать их в любом случае, — далеко не сотни мегабайт. Это неудивительно, поскольку рабочая нагрузка на этом сервере почти полностью состоит из запросов `SELECT`. Следовательно, мы можем сделать вывод: вместо того чтобы обвинять проблему в сбросе InnoDB, мы должны обвинить в проблеме задержку сброса InnoDB. Это симптом, результат, а не причина. Лежащая в основе проблема приводит к тому, что диски заполняются настолько, что InnoDB не успевает выполнить свои задачи ввода/вывода. Поэтому мы можем отбросить это как возможную причину и зачеркнуть одну из наших интуитивных идей.

Иногда довольно сложно отличить причину от результата, и когда проблема выглядит знакомой, может возникнуть соблазн пропустить анализ и перейти к диагнозу. Срезать углы не стоит, но в то же время важно прислушиваться к своей интуиции. Если что-то кажется знакомым, разумно потратить немного времени на измерение необходимых и достаточных условий, чтобы понять, является ли это проблемой. Это может сэкономить много времени, которое вы потратили бы на получение других данных о системе и ее производительности. Просто постарайтесь не делать выводов, основанных на интуиции, наподобие: «Я видел это раньше и уверен, что это то же самое». Если можете, соберите доказательства.

Следующим шагом стала попытка выяснить, что вызывало такое странное использование ввода/вывода сервера. Мы обращаем ваше внимание на приведенные ранее рассуждения: «Сервер пишет сотни мегабайт на диск в течение многих секунд, но база данных содержит только 900 Мбайт. Как это могло произойти?» Обратили ли

[1] Или как там эта фраза звучит? Положить все яйца в один стог сена?

вы внимание на неявное предположение, что база данных выполняет запись? Какими мы располагаем доказательствами, что это именно база данных? Попытайтесь остановиться, если принимаете что-то на веру бездоказательно, и если что-то не имеет смысла, подумайте, нет ли тут предположений. Если возможно, выполните измерения и устраните все сомнения.

Мы видели два варианта. Либо база данных вызывала работу ввода/вывода, и в этом случае, если бы мы могли понять, почему так происходит, мы считали бы, что это приведет нас к причине проблемы. Либо база данных не выполняла весь этот ввод/вывод, но инициатором этих операций было что-то еще, и нехватка ресурсов ввода/вывода могла повлиять на базу данных. Мы говорим об этом очень осторожно, чтобы избежать другого неявного предположения: занятость дисков не означает, что MySQL пострадает. Помните, что основная рабочая нагрузка этого сервера — чтение из оперативной памяти, поэтому вполне можно представить, что диски могли бы перестать отвечать на запросы в течение длительного времени, не вызывая серьезных проблем.

Если вы следите за нашими рассуждениями, то, возможно, видите, что нам нужно вернуться назад и проверить другое предположение. Мы видим, что дисковое устройство ведет себя неправильно, о чем свидетельствует большое время ожидания. Твердотельный накопитель не должен тратить в среднем четверть секунды на каждую операцию ввода/вывода. И действительно, мы видим, что, по данным iostat, сам диск реагирует быстро, но много времени затрачивается на проход через очередь блочного устройства к диску. Помните, что это только предположение iostat, — и это может быть ошибкой.

В чем причина плохой производительности

Когда ресурс неправильно работает, полезно попытаться понять, почему так происходит. Существует несколько причин.

1. Ресурс перегружен работой, и ему не хватает мощностей для правильной работы.
2 Ресурс не настроен должным образом.
3 Ресурс сломан или неисправен.

В анализируемом кейсе вывод утилиты iostat может указывать либо на слишком большой объем работы, либо на неправильную конфигурацию (почему требования ввода/вывода так долго стоят в очереди, прежде чем попасть на диск, если на самом деле он реагирует быстро?). Однако важным элементом поиска ошибки является сравнение спроса на систему с ее мощностью. Из результатов экстенсивного эталонного тестирования мы знаем, что конкретный накопитель SSD, используемый этим клиентом, не может выдерживать сотни мегабайт записей в секунду. Таким образом, хотя iostat утверждает, что диск отвечает очень хорошо, вполне вероятно, что это не совсем так. В данном случае мы никак не могли доказать, что диск был медленнее, чем утверждала iostat, но это выглядело весьма вероятным. Тем не менее это не меняет нашего мнения: причиной могут быть неверное использование диска, неверная настройка или и то и другое.

После работы с данными диагностики, позволившими сделать такой вывод, следующая задача была очевидна: измерить то, что вызывает ввод/вывод. К сожалению, это было неосуществимо в используемой клиентом версии GNU/Linux. Постаравшись, мы могли бы получить обоснованное предположение, но сначала хотели рассмотреть другие варианты. В качестве альтернативного варианта мы могли бы определить, сколько операций ввода/вывода поступает из MySQL, но опять же в этой версии MySQL данная операция была неосуществима из-за отсутствия необходимых инструментов.

Вместо этого мы решили попробовать понаблюдать за вводом/выводом MySQL, основываясь на знании о том, как система использует диск. В основном MySQL записывает на диск только данные, журналы, файлы сортировки и временные таблицы. Мы исключили данные и журналы из рассмотрения на основе счетчиков состояния и другой информации, о которой говорили ранее. Теперь предположим, что MySQL внезапно запишет на диск кучу данных из временных таблиц или файлов сортировки. Как можно за этим понаблюдать? Существует два простых способа: оценить объем свободного места на диске или посмотреть открытые дескрипторы сервера с помощью команды `lsof`. Мы сделали и то и другое, и результаты нас вполне удовлетворили. Приведем то, что `df -h` показывало каждую секунду во время инцидента, который мы изучали:

```
Filesystem  Size  Used  Avail  Use%  Mounted  on
/dev/sda3   58G   20G   36G    36%   /
/dev/sda3   58G   20G   36G    36%   /
/dev/sda3   58G   19G   36G    35%   /
/dev/sda3   58G   19G   36G    35%   /
/dev/sda3   58G   19G   36G    35%   /
/dev/sda3   58G   19G   36G    35%   /
/dev/sda3   58G   18G   37G    33%   /
/dev/sda3   58G   18G   37G    33%   /
/dev/sda3   58G   18G   37G    33%   /
```

Далее представлены данные, полученные с помощью команды `lsof`, которые мы почему-то собирали только каждые 5 секунд. Мы просто суммируем размеры всех файлов, которые mysqld открывает в `/tmp`, и распечатываем сумму для каждой временной выборки в файле:

```
$ awk '
   /mysqld.*tmp/ {
     total += $7;
   }
   /^Sun Mar 28/ && total {
     printf "%s %7.2f MB\n", $4, total/1024/1024;
     total = 0;
   }' lsof.txt
18:34:38 1655.21 MB
18:34:43    1.88 MB
18:34:48    1.88 MB
18:34:53    1.88 MB
18:34:58    1.88 MB
```

На основе полученной информации можно сделать вывод, что MySQL записывает около 1,5 Гбайт данных во временные таблицы на начальных этапах инцидента, а это

соответствует тому, что мы обнаружили в состояниях `SHOW PROCESSLIST` («Копирование в таблицу tmp»). Свидетельства указывают на массу плохих запросов, которые мгновенно заполняют диск. Самая распространенная причина этого, которую мы видели (сработала наша интуиция), — паника в кэше, когда все кэшированные объекты разом утекают из memcached, а несколько приложений дружно пытаются повторно восстановить кэш. Мы показали выборку запросов разработчикам и обсудили их цель. В самом деле оказалось, что причиной было одновременное прекращение существования кэша (интуитивное предположение подтвердилось). Разработчики попытались решить эту проблему на уровне приложения, а мы помогли им изменить запросы так, чтобы они не использовали временные таблицы на диске. Даже одно из этих исправлений могло бы предотвратить возникновение проблемы, но лучше внести их оба.

Теперь мы хотели бы вернуться немного назад и ответить на вопросы, которые могли у вас возникнуть (описывая в этой главе свой подход, мы критиковали его).

- *Почему для начала мы просто не оптимизировали медленные запросы?*

  Потому что проблема была не в медленных запросах, это была ошибка типа «слишком много соединений». Конечно, логично ожидать, что длительные запросы приводят к накоплению запросов и увеличению количества подключений. Но к такому результату могут привести и десятки других событий. Если не найдена точная причина, объясняющая, почему все идет не так, возникает соблазн вернуться к поиску медленных запросов или других общих моментов, которые выглядят так, как будто могут быть улучшены[1]. К сожалению, чаще такой подход приводит к негативным результатам. Если вы отвезли свой автомобиль к механику и пожаловались на непонятный шум, а затем были огорошены счетом за балансировку шин и смену трансмиссионной жидкости, потому что механик не смог найти настоящую проблему и начал искать другие, разве вы не будете в ярости?

- *Но разве не сигнал тревоги то, что запросы выполняются медленно с плохим EXPLAIN?*

  Они действительно выполнялись медленно, но во время инцидентов. Были ли это причина или результат? Это не было понятно, пока мы не углубились в исследование. Не забывайте, что в обычных условиях запросы выполнялись довольно хорошо. Просто то, что запрос делает файловую сортировку[2] с помощью времен-

---

[1] Этот подход также известен в формулировке «Когда у вас есть только молоток, все вокруг похоже на гвоздь».

[2] В статье «Что означает использование файловой сортировки в MySQL» один из авторов этой книги, Бэрон Шварц (Baron Schwartz), пишет: «Файловая сортировка — это плохое название. Каждый раз, когда сортировку нельзя выполнить с помощью индекса, она становится файловой. При этом никакого отношения к файлам она не имеет. Ее следовало назвать просто сортировкой». — *Примеч. пер.*

ной таблицы, не означает, что это проблема. Избавление от файловой сортировки и временных таблиц — это обычная тактика хорошего разработчика.

Но не всегда обычные тактики помогают в решении специфических проблем. Например, проблемой могла быть неправильная конфигурация. Мы видели немало случаев, когда некто пытался исправить неверно сконфигурированный сервер оптимизацией запросов, что оказывалось пустой тратой времени и просто увеличивало ущерб от реальной проблемы.

- *Если кэшированные элементы неоднократно восстанавливались, не возникало ли множества одинаковых запросов?*

  Да, и это то, что мы тогда не проверили. Множественные потоки, восстанавливающие один и тот же элемент в кэше, действительно вызывали бы возникновение множества полностью идентичных запросов. (Это не то же самое, что несколько запросов одного и того же общего типа, которые, например, могут различаться параметром в разделе `WHERE`.) Если бы мы обратили на это внимание, то могли бы, прислушавшись к интуиции, быстрее прийти к решению.

- *Выполнялись сотни запросов SELECT в секунду, но только пять UPDATE. Можно ли сказать, что эти пять не были действительно тяжелыми запросами?*

  Они действительно могли сильно нагружать сервер. Мы не показывали фактические запросы, поскольку эта информация лишь помешала бы, но мнение, что абсолютное число запросов каждого типа не обязательно важно, имеет право на жизнь.

- *Разве «доказательство», говорящее об источнике урагана ввода/вывода, все еще недостаточно весомое?*

  Да. Может быть много объяснений, почему небольшая база данных записывает огромное количество данных на диск или почему свободное место на диске быстро сокращается. Это то, что довольно сложно измерить (хотя и не невозможно) в версиях MySQL и GNU/Linux. Хотя можно сыграть в адвоката дьявола и придумать множество сценариев. Мы решили уравновесить затраты и потенциальную выгоду, сначала выбрав наиболее перспективный путь. Чем сложнее проведение измерений и меньше уверенность в результатах, тем выше соотношение затрат и выгод и тем сильнее мы готовы принять неопределенность.

- *Мы говорили: «Раньше база данных никогда не была проблемой». Разве это не предубеждение?*

  Да, это было необъективно. Если вы уловили это — здорово, если нет, то, надеемся, это станет хорошим примером того, что у всех нас есть предубеждения.

Мы хотели бы завершить кейс по устранению неполадок, указав, что рассмотренную проблему, скорее всего, можно было бы решить (или предотвратить) без нашего участия, используя инструмент профилирования приложений, например New Relic.

Другие инструменты профилирования

Мы показали множество способов профилирования MySQL, операционной системы и запросов. Продемонстрировали те, которые, на наш взгляд, являются наиболее полезными, и, конечно же, покажем еще больше инструментов и методик для проверки и измерения систем в этой книге. Но подождите, это еще не все!

Таблицы USER_STATISTICS

Пакеты Percona Server и MariaDB включают дополнительные таблицы `INFORMATION_SCHEMA` для получения статистики использования на уровне объекта. Они были созданы в Google. Такие информационные таблицы чрезвычайно полезны для определения того, в какой степени используются различные части сервера. На крупном предприятии, где администраторы баз данных отвечают за управление базами данных и слабо контролируют разработчиков, подобные таблицы могут быть жизненно важными для измерения и аудита активности базы данных и обеспечения соблюдения политики использования. Они также полезны для многоклиентских приложений, таких как среда совместного хостинга. В то же время, когда вы ищете источник проблем с производительностью, они могут помочь выяснить, кто максимально задействует базу данных или какие таблицы и индексы наиболее часто или редко используются. Рассмотрим эти таблицы:

```
mysql> SHOW TABLES FROM INFORMATION_SCHEMA LIKE '%_STATISTICS';
+-------------------------------------------+
| Tables_in_information_schema (%_STATISTICS) |
+-------------------------------------------+
| CLIENT_STATISTICS                         |
| INDEX_STATISTICS                          |
| TABLE_STATISTICS                          |
| THREAD_STATISTICS                         |
| USER_STATISTICS                           |
+-------------------------------------------+
```

В книге не хватит места для примеров всех запросов, которые вы можете выполнить к этим таблицам, но несколько пунктов привести не помешает.

- Вы можете найти наиболее и наименее используемые таблицы и индексы для чтения, обновления или того и другого.
- Вы можете найти неиспользуемые индексы, которые станут кандидатами на удаление.
- Вы можете сравнить `CONNECTED_TIME` и `BUSY_TIME` подчиненного сервера, чтобы понять, ожидается ли в ближайшем будущем активная работа по репликации данных.

В MySQL 5.6 Performance Schema добавляет таблицы, которые служат для достижения аналогичных целей.

Инструмент strace

Инструмент strace перехватывает системные вызовы. Существует несколько способов его использования. Один из них — фиксация продолжительности времени вызова системы и распечатка профиля:

```
$ strace -cfp $(pidof mysqld)
Process 12648 attached with 17 threads - interrupt to quit
^CProcess 12648 detached
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ------------
 73.51    0.608908       13839        44           select
 24.38    0.201969       20197        10           futex
  0.76    0.006313           1     11233         3 read
  0.60    0.004999         625         8           unlink
  0.48    0.003969          22       180           write
  0.23    0.001870          11       178           pread64
  0.04    0.000304           0      5538           _llseek
[несколько строк опущены для краткости]
------ ----------- ----------- --------- --------- ------------
100.00    0.828375                17834        46 total
```

Таким образом, этот инструмент немного похож на oprofile. Однако последний будет профилировать внутренние символы программы, а не только системные вызовы. Кроме того, strace использует другую методику перехвата вызовов, которая немного непредсказуема и значительно увеличивает издержки. К тому же strace измеряет время, тогда как oprofile определяет, где расходуются циклы процессора. В качестве примера: strace позволяет обнаружить случаи, когда ожидания ввода/вывода доставляют проблемы, поскольку strace выполняет измерения от начала и до конца системных вызовов, таких как `read` или `pread64`, а oprofile – нет, поскольку системные вызовы ввода/вывода фактически не загружают процессор, а просто ожидают завершения этого ввода/вывода.

Мы используем oprofile только по необходимости, поскольку при большом многопоточном процессе, таком как mysqld, у него могут появиться странные побочные эффекты, и в случае применения инструмента strace mysqld будет работать так медленно, что окажется практически непригодным для рабочей нагрузки. Тем не менее в некоторых ситуациях он может быть чрезвычайно полезен, а в пакете Percona Toolkit есть инструмент pt-ioprofile, который использует strace для генерации истинного профиля активности ввода/вывода. Это может пригодиться для доказательства или опровержения разных трудноизмеримых событий, которые в противном случае мы не смогли бы завершить. (Если сервер работает с MySQL 5.6, вместо этого мы могли бы использовать Performance Schema.)

Итоги главы

Эта глава закладывает основы хода и методики мышления, необходимых для успешной оптимизации производительности. Правильный подход — ключ к полному рас-

крытию потенциала разрабатываемых вами систем и применению знаний, которые вы приобретете, прочитав следующие главы книги. Перечислим основополагающие принципы, которые мы попытались проиллюстрировать.

- Мы считаем, что самым правильным способом определения производительности является время отклика системы.
- Вы не можете гарантированно улучшить то, что нельзя измерить, поэтому увеличение производительности лучше всего работает с высококачественными, правильно выполненными, полными измерениями времени отклика.
- Лучшим местом для начала измерений является приложение, а не база данных. Если на нижних уровнях есть проблема, например, с базой данных, она проявится в результате правильно выполненных измерений.
- Большинство систем невозможно измерить полностью, и измерения всегда ошибочны. Но даже в этом случае, признав несовершенство и неточность используемых средств, обычно можно обойти ограничения и получить хороший результат.
- В результате тщательных измерений появляется слишком много данных для анализа, поэтому вам нужен профиль. Это лучший инструмент для подъема важнейших данных на самый верх. Тем самым мы можем решить, с чего следует начинать.
- Профиль — это сводка, которая скрывает и отбрасывает детали. Но он и не показывает вам, чего не хватает. Не стоит рассматривать профиль как истину в последней инстанции.
- Существует два вида потребления времени: работа и ожидание. Многие инструменты профилирования могут только измерять работу, поэтому анализ ожиданий иногда является важным дополнением, особенно когда уровень задействования процессора низок, а работа все еще не завершена.
- Оптимизация — это не то же самое, что улучшение. Прекратите работать, если дальнейшее улучшение не стоит затрат на него.
- Прислушайтесь к своей интуиции, но используйте ее для анализа, а не для принятия решений об изменениях в системе. Максимально основывайте свои решения на данных, а не на ощущениях.

Общий подход, который мы продемонстрировали, решая проблемы с производительностью, состоит в том, что сначала следует прояснить вопрос, а затем выбрать подходящую методику для ответа на него. Если вы пытаетесь понять, можно ли улучшить производительность сервера в целом, то начните с журналирования всех запросов и создания общесистемного профиля с помощью утилиты pt-query-digest. Если вы ищете неправильные запросы, о которых, возможно, и не знаете, журналирование и профилирование также могут помочь. Посмотрите на процессы, потребляющие больше всего времени, запросы, доставляющие пользователям больше всего неприятных моментов при работе с системой, и запросы, которые сильно изменяются или имеют странные гистограммы времени отклика. Обнаружив плохие запросы, углубитесь в них, просмотрев подробный отчет с помощью pt-query-digest, используя команду `SHOW PROFILE` и аналогичные инструменты, например `EXPLAIN`.

Если запросы выполняются плохо по какой-либо явной причине, возможно, вы столкнулись со спорадическими проблемами сервера. Чтобы выяснить это, вы можете измерить и отобразить на графике счетчики состояния сервера на хорошем уровне детализации. Если это позволит выявить проблему, используйте те же самые данные для формулировки надежного условия триггера, чтобы иметь возможность фиксировать пакет подробных диагностических данных. Затратьте столько времени и сил, сколько необходимо для нахождения хорошего условия триггера, которое позволит избежать ложноположительных и ложноотрицательных результатов. Если вы зафиксировали проблему в момент появления, но не можете определить ее причину, собирайте дополнительные данные или попросите о помощи.

Вы работаете с системами, которые не можете измерить полностью, но это просто конечные автоматы, поэтому если вы будете внимательны, логичны и настойчивы, то, скорее всего, получите результаты. Старайтесь не путать результаты с причинами и не вносить изменения, не определив проблему.

Теоретически оптимальный подход с вертикальным профилированием и всеобъемлющими измерениями — идеал, к которому стоит стремиться, но чаще нам приходится иметь дело с реальными системами. Реальные системы сложны и недостаточно хорошо оснащены инструментами, поэтому мы делаем все возможное с помощью того, что имеем. Такие инструменты, как pt-query-digest и анализатор запросов MySQL Enterprise Monitor, неидеальны и часто не показывают достоверно источник проблемы. Но в основном они вполне справляются со своими задачами.

4 Оптимизация схемы и типов данных

Высокой производительности можно достичь только при хорошем логическом и физическом проектировании схемы. Следует проектировать свою схему для конкретных запросов, которые вы будете запускать. При этом часто приходится идти на компромиссы. Например, денормализованная схема может ускорить одни типы запросов, но замедлить другие. Добавление таблиц счетчиков и сводных таблиц — хороший способ оптимизации запросов, но их поддержка может дорого стоить. Особенности использования MySQL незначительно влияют на это.

Текущая глава и глава 5, посвященная индексированию, охватывают характерные для MySQL области проектирования схемы. При этом мы предполагаем, что вы знаете, как проектировать базы данных. Поскольку это глава по проектированию базы данных MySQL, речь в ней пойдет о различиях при проектировании баз данных в MySQL и других СУРБД. Если вы хотите изучить основы проектирования баз данных, предлагаем прочесть книгу Клер Чурчер (Clare Churcher) *Beginning Database Design* (издательство Apress).

Данная глава подготовит вас к прочтению двух следующих. В главах 4–6 мы рассмотрим взаимодействие логического и физического проектирования и выполнения запросов. Для этого потребуется как взгляд на картину в целом, так и внимание к мелочам. Вам нужно разобраться в системе, чтобы понять, как одна часть повлияет на другие. Возможно, полезно будет перечитать эту главу после изучения разделов, посвященных индексированию и оптимизации запросов. Большинство обсуждаемых тем нельзя рассматривать изолированно.

Выбор оптимальных типов данных

MySQL поддерживает множество типов данных. Выбор правильного типа для хранения вашей информации критичен с точки зрения увеличения производительности. Следующие простые рекомендации помогут сделать правильный выбор независимо от типа сохраняемых данных.

- *Меньше — обычно лучше.* Старайтесь использовать типы данных минимального размера, достаточного для их правильного хранения и представления. Меньшие по размеру типы данных обычно быстрее, поскольку занимают меньше места на диске, в памяти и кэше процессора. Кроме того, для их обработки обычно требуется меньше процессорного времени.

  Однако убедитесь, что правильно представляете диапазон возможных значений данных, поскольку увеличение размерности типа данных на многих участках схемы может оказаться болезненным и длительным процессом. Если вы сомневаетесь в том, какой тип данных использовать, выберите самый короткий при условии, что его размера будет достаточно. (Если система не сильно загружена, хранит не очень много данных или вы находитесь на раннем этапе процесса проектирования, то легко сможете позже изменить решение.)

- *Просто — значит хорошо.* Для выполнения операций с более простыми типами данных обычно требуется меньше процессорного времени. Например, сравнение целых чисел менее затратно, чем сравнение символов, поскольку различные кодировки и схемы упорядочения (правила сортировки) усложняют сравнение символов. Приведем два примера: значения даты и времени следует хранить во встроенных типах данных MySQL, а не в строках. Для IP-адресов используйте целочисленные типы данных. Мы обсудим этот вопрос позднее.

- *Насколько это возможно, избегайте значений NULL.* Очень часто в таблицах встречаются поля, допускающие хранение `NULL` (отсутствие значения), хотя приложению это совершенно не нужно просто потому, что такой режим установлен по умолчанию. Как правило, стоит объявить столбец как `NOT NULL`, если только вы не планируете хранить в нем значения `NULL`.

  MySQL тяжело оптимизировать запросы, содержащие допускающие `NULL` столбцы, поскольку из-за них усложняются индексы, статистика индексов и сравнение значений. Столбец, допускающий `NULL`, занимает больше места на диске и требует специальной обработки. Когда такой столбец проиндексирован, ему требуется дополнительный байт для каждой записи, а в MyISAM даже может возникнуть ситуация, когда придется преобразовать индекс фиксированного размера (например, индекс по одному целочисленному столбцу) в индекс переменного размера.

  Повышение производительности в результате замены столбцов `NULL` на `NOT NULL` обычно невелико, так что не делайте их поиск и изменение в существующих схемах приоритетными, если не уверены, что именно они вызывают проблемы. Но если вы планируете индексировать столбцы, по возможности не делайте их допускающими `NULL`.

  Конечно, бывают и исключения. Например, стоит упомянуть, что подсистема InnoDB использует для хранения значения `NULL` один бит, поэтому она может быть довольно экономичной для неплотно заполненных данных. Однако это не относится к MyISAM.

Первым шагом при выборе типа данных для столбца является определение общего класса типов: числовые, строковые, временные и т. п. Обычно никаких сложностей при этом не возникает, однако бывают ситуации, когда выбор неочевиден.

Следующим шагом является выбор конкретного типа. Многие типы данных MySQL позволяют хранить данные одного и того же вида, но с разными диапазоном значений и точностью или требующие разного физического пространства (на диске или в памяти). Кроме того, некоторые типы данных обладают специальным поведением или свойствами.

Например, в столбцах `DATETIME` и `TIMESTAMP` можно хранить один и тот же тип данных: дату и время с точностью до секунды. Однако тип `TIMESTAMP` требует вдвое меньше места, позволяет работать с часовыми поясами и обладает специальными средствами автоматического обновления. При этом диапазон допустимых значений для него намного у́же, а специальные средства могут стать недостатком.

Здесь мы обсудим основные типы данных. В целях совместимости MySQL поддерживает различные псевдонимы, например `INTEGER`, `BOOL` и `NUMERIC`. Все это именно псевдонимы. Данный факт может сбить с толку, но не влияет на производительность. Если вы создадите таблицу с псевдонимом типа данных, а затем запустите команду `SHOW CREATE TABLE`, то увидите, что MySQL выдает базовый тип, а не использованный псевдоним.

Целые числа

Существуют два типа чисел: целые и вещественные (числа с дробной частью). Для хранения целых чисел используйте один из целочисленных типов: `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT` или `BIGINT`. Для хранения они требуют 8, 16, 24, 32 и 64 бита соответственно. Они позволяют хранить значения в диапазоне от $-2(N - 1)$ до $2(N - 1) - 1$, где N — количество битов, использованных для их хранения.

Целые типы данных могут иметь необязательный атрибут `UNSIGNED`, запрещающий отрицательные значения и приблизительно вдвое увеличивающий верхний предел хранимых положительных значений. Например, тип `TINYINT UNSIGNED` позволяет хранить значения от 0 до 255, а не от −128 до 127.

Знаковые и беззнаковые типы требуют одинакового пространства и обладают одинаковой производительностью, так что используйте тот тип, который больше подходит для диапазона ваших данных.

Ваш выбор определяет то, как СУБД MySQL хранит данные в памяти и на диске. Однако для *целочисленных вычислений* обычно используются 64-разрядные целые типа `BIGINT`, даже на машинах с 32-разрядной архитектурой (исключение составляют некоторые агрегатные функции, использующие для вычислений тип `DECIMAL` или `DOUBLE`).

MySQL позволяет указать для целых чисел размер, например `INT(11)`. Для большинства приложений это неважно — это не ограничивает диапазон возможных значений, лишь определяет, сколько позиций интерактивным инструментам MySQL (напри-

мер, клиенту командной строки) необходимо зарезервировать для вывода числа. С точки зрения хранения и вычисления `INT(1)` и `INT(20)` идентичны.

Подсистемы хранения сторонних производителей, например Infobright, иногда используют собственный формат хранения данных и схемы сжатия. При этом они могут отличаться от встроенных в MySQL подсистем хранения.

Вещественные числа

Вещественные числа — это числа, имеющие дробную часть. Однако они используются не только для дробных чисел — в типе данных `DECIMAL` также можно хранить большие числа, не помещающиеся в типе `BIGINT`. MySQL поддерживает как «точные», так и «неточные» типы.

Типы `PLOAT` и `DOUBLE` допускают приближенные математические вычисления с плавающей точкой. Если вам нужно точно знать, как вычисляются результаты с плавающей точкой, изучите, как это реализовано на имеющейся у вас платформе.

Тип `DECIMAL` используется для хранения точных дробных чисел. В MySQL версии 5.0 и более поздних `DECIMAL` поддерживает еще и точные вычисления. MySQL 4.1 и более ранние версии для работы с `DECIMAL` применяли вычисления с плавающей точкой, которые иногда давали странные результаты из-за ухудшения точности. В этих версиях MySQL тип `DECIMAL` предназначался только для хранения.

Начиная с версии MySQL 5.0 математические операции с типом `DECIMAL` реализуются самим сервером баз данных, поскольку непосредственно процессоры не поддерживают такие вычисления. Операции с плавающей точкой протекают существенно быстрее, так как их выполнение для процессора естественно.

Как типы с плавающей запятой, так и тип `DECIMAL` позволяют задавать нужную точность. Для столбца типа `DECIMAL` вы можете определить максимальное количество цифр до и после запятой. Это влияет на объем пространства, требующегося для хранения данных столбца. MySQL 5.0 и более новые версии упаковывают цифры в двоичную строку (девять цифр занимают четыре байта). Например, `DECIMAL(18, 9)` будет хранить девять цифр с каждой стороны от запятой, используя в общей сложности 9 байт: 4 байта для цифр перед запятой, 1 байт для самой запятой и 4 байта для цифр после нее.

Число типа `DECIMAL` в MySQL 5.0 и более новых версиях может содержать до 65 цифр. Более ранние версии MySQL имели предел 254 цифры и хранили значения в виде неупакованных строк (1 байт на цифру). Однако эти версии MySQL на деле не могли использовать такие большие числа в вычислениях, поскольку тип `DECIMAL` был просто форматом хранения. При выполнении вычислений значения `DECIMAL` преобразовывались в тип `DOUBLE`.

Вы можете указать желаемую точность для столбца чисел с плавающей точкой несколькими способами так, что MySQL незаметно для пользователя выберет другой тип данных или при сохранении будет округлять значения. Эти спецификаторы точности нестандартны, поэтому мы рекомендуем задавать желаемый тип, но не точность.

Типы с плавающей точкой обычно задействуют для хранения одного и того же диапазона значений меньше пространства, чем тип `DECIMAL`. Столбец типа `FLOAT` использует 4 байта. Тип `DOUBLE` требует 8 байт и имеет бóльшую точность и больший диапазон значений, чем `FLOAT`. Как и в случае работы с целыми числами, вы выбираете тип только для хранения. MySQL использует тип `DOUBLE` для вычислений с плавающей точкой.

Из-за дополнительных требований к пространству и затрат на вычисления тип `DECIMAL` стоит использовать только тогда, когда нужны точные результаты при вычислениях с дробными числами, например при хранении финансовых данных.

Но при некоторых операциях с большими числами целесообразнее применять тип `BIGINT` вместо `DECIMAL` и хранить данные как кратные наименьшей доле валюты, которую вам нужно обрабатывать. Предположим, необходимо хранить финансовые данные с точностью до 1/10 000 цента. Вы можете умножить все суммы в долларах на 1 миллион и сохранить результат в `BIGINT`. Тем самым избежите как неточности хранения типов с плавающей запятой, так и высоких затрат на точные расчеты типа `DECIMAL`.

Строковые типы

MySQL поддерживает большой диапазон строковых типов данных с различными вариациями. Эти типы претерпели значительные изменения в версиях 4.1 и 5.0, что сделало их еще более сложными. Начиная с версии MySQL 4.1, каждый строковый столбец может иметь собственную кодировку и соответствующую схему упорядочения (более подробно эта тема рассматривается в главе 7). Данное обстоятельство может значительно повлиять на производительность.

Типы VARCHAR и CHAR

Два основных строковых типа — это `VARCHAR` и `CHAR`, предназначенные для хранения символьных значений. К сожалению, нелегко объяснить, как именно эти значения хранятся в памяти и на диске, поскольку конкретная реализация зависит от выбранной подсистемы хранения. Мы предполагаем, что вы используете InnoDB и/или MyISAM. В противном случае вам лучше прочитать документацию по используемой подсистеме хранения.

Давайте посмотрим, как обычно значения `VARCHAR` и `CHAR` сохраняются на диске. Имейте в виду, что подсистема хранения может хранить `CHAR` или `VARCHAR` в памяти не так, как на диске, и что сервер может преобразовывать данные в другой формат,

когда извлекает их из подсистемы хранения. Приведем общее сравнение этих двух типов.

- VARCHAR. Тип VARCHAR хранит символьные строки переменной длины и является наиболее используемым строковым типом данных. Строки этого типа могут занимать меньше места, чем строки фиксированной длины, поскольку для VARCHAR используется столько места, сколько действительно необходимо (то есть для хранения более коротких строк требуется меньше пространства). Исключением являются таблицы типа MyISAM, созданные с параметром ROW_FORMAT=FIXED. В этом случае для каждой строки на диске отводится область фиксированного размера, поэтому место может расходоваться впустую.

  В типе VARCHAR для хранения длины строки используются 1 или 2 дополнительных байта: 1 байт, если максимальная длина строки в столбце не превышает 255 байт, и 2 байта — при более длинных строках. Если используется кодировка latin1, тип VARCHAR(10) может занимать до 11 байт. Тип VARCHAR(1000) может использовать до 1002 байт, поскольку 2 байта требуется для хранения информации о длине строки.

  Тип VARCHAR увеличивает производительность за счет экономии места. Но поскольку это строки переменной длины, они способны увеличиваться при обновлении, что требует дополнительной работы. Если строка становится длиннее и больше не помещается в отведенное для нее место, то дальнейшее поведение системы зависит от подсистемы хранения. Например, MyISAM может фрагментировать строку, а InnoDB, возможно, придется выполнить разбиение страницы. Другие подсистемы хранения могут вообще не обновлять данные в месте их хранения.

  Обычно целесообразно использовать тип VARCHAR, если максимальная длина строки в столбце значительно больше средней, обновление поля выполняется редко, так что фрагментация не представляет проблемы, и если применяется сложная кодировка, например UTF-8, в которой для хранения одного символа используется переменное количество байтов.

  В MySQL 5.0 и более новых версиях при сохранении и извлечении текстовых строк MySQL сохраняет пробелы в конце строки. В версии же 4.1 и более ранних они удаляются.

  Интереснее ситуация с InnoDB, которая может хранить длинные значения типа VARCHAR в виде BLOB. Подробнее мы рассмотрим эту способность позже.

- CHAR. Тип CHAR имеет фиксированную длину: MySQL всегда выделяет место для указанного количества символов. Сохраняя значения CHAR, MySQL удаляет все пробелы в конце строки (это справедливо также для типа VARCHAR в MySQL 4.1 и более ранних версий — CHAR и VARCHAR были логически идентичны и различались только форматом хранения). К значениям для сравнения добавляются пробелы.

  Тип CHAR полезен, когда требуется сохранять очень короткие строки или все значения имеют приблизительно одинаковую длину. Например, CHAR хорошо подходит для хранения MD5-сверток паролей пользователей, которые всегда имеют одинаковую длину. Также тип CHAR имеет преимущество над VARCHAR при

часто меняющихся данных, поскольку строка фиксированной длины не подвержена фрагментации. Если в столбцах очень короткие строки, тип CHAR также эффективнее, чем VARCHAR: если тип CHAR(1) применяется для хранения значений только Y и N, то в однобайтовой кодировке[1] он займет лишь 1 байт, тогда как для типа VARCHAR(1) из-за наличия дополнительного байта длины строки потребуется 2 байта.

Это поведение может несколько сбивать с толку, поэтому проиллюстрируем его на примере. Сначала мы создали таблицу с одним столбцом типа CHAR(10) и сохранили в нем какие-то значения:

```
mysql> CREATE TABLE char_test( char_col CHAR(10));
mysql> INSERT INTO char_test(char_col) VALUES
    -> ('string1'), (' string2'), ('string3 ');
```

При извлечении значений пробелы в конце строки удаляются:

```
mysql> SELECT CONCAT("'", char_col, "'") FROM char_test;
+---------------------------+
| CONCAT("'", char_col, "'") |
+---------------------------+
| 'string1'                 |
| '  string2'               |
| 'string3'                 |
+---------------------------+
```

Если мы сохраним те же значения в столбце типа VARCHAR(10), то после извлечения получим следующий результат:

```
mysql> SELECT CONCAT("'", varchar_col,"'") FROM varchar_test;
+---------------------------+
| CONCAT("'", char_col, "'") |
+---------------------------+
| 'string1'                 |
| '  string2'               |
| 'string3 '                |
+---------------------------+
```

Как именно записываются данные, зависит от подсистемы хранения, и не все подсистемы одинаково обрабатывают значения фиксированной и переменной длины. В подсистеме Memory используются строки фиксированной длины, поэтому ей приходится выделять максимально возможное место для каждого значения, даже если это поле переменной длины[2]. Однако поведение при дополнении и удалении пробелов одинаково для всех подсистем хранения, поскольку эту работу выполняет сам сервер MySQL.

Родственными типами для CHAR и VARCHAR являются BINARY и VARBINARY, предназначенные для хранения двоичных строк. Двоичные строки очень похожи на обычные, но вместо символов в них хранятся байты. Метод дополнения пробелами также от-

[1] Помните, что длина указана в символах, а не в байтах. При многобайтовой кодировке для хранения каждого символа может потребоваться более 1 байта.

[2] Подсистема Memory в пакете Percona Server поддерживает строки переменной длины.

личается: MySQL добавляет в строки типа **BINARY** значение **\0** (нулевой байт) вместо пробелов и не удаляет дополненные байты при извлечении[1].

Эти типы полезны, когда нужно хранить двоичные данные и вы хотите, чтобы MySQL сравнивала значение как байты, а не как символы. Преимуществом побайтового сравнения является не только нечувствительность к регистру. MySQL сравнивает строки **BINARY** побайтно в соответствии с числовым значением каждого байта. Так что двоичное сравнение может оказаться значительно проще символьного и, как следствие, быстрее.

Щедрость не всегда разумна

Значение **'hello'** занимает одно и то же место и в столбце типа **VARCHAR(5)**, и в столбце типа **VARCHAR(200)**. Обеспечивает ли преимущество применение более короткого столбца?

Оказывается, преимущество есть, и значительное. Для большего столбца может потребоваться намного больше памяти, поскольку MySQL часто выделяет для внутреннего хранения значений участки памяти фиксированного размера. Это особенно плохо для сортировки или операций, использующих временные таблицы в памяти. То же самое происходит при файловой сортировке, использующей временные таблицы на диске.

Наилучшей стратегией будет выделение такого объема памяти, который действительно нужен.

Типы BLOB и TEXT

Строковые типы **BLOB** и **TEXT** предназначены для хранения больших объемов двоичных или символьных данных соответственно.

Фактически это семейства типов данных: к символьным типам относятся **TINYTEXT**, **SMALLTEXT**, **TEXT**, **MEDIUMTEXT** и **LONGTEXT**, а к двоичным — **TINYBLOB**, **SMALLBLOB**, **BLOB**, **MEDIUMBLOB** и **LONGBLOB**. **BLOB** является синонимом для **SMALLBLOB**, а **TEXT** — для **SMALLTEXT**.

В отличие от остальных типов данных каждое значение **BLOB** и **TEXT** MySQL обрабатывает как отдельный объект. Подсистемы хранения часто хранят их особым образом: если они большого размера, InnoDB может использовать для них отдельную внешнюю область хранения. Каждому значению такого типа требуется от 1 до 4 байт в самой строке и достаточно места во внешнем хранилище для хранения фактического значения.

Единственное различие между семействами **BLOB** и **TEXT** заключается в том, что типы **BLOB** хранят двоичные данные без учета схемы упорядочения и кодировки, а типы **TEXT** используют схемы упорядочения и кодировку.

[1] Будьте осторожны с типом BINARY, если значение после извлечения должно оставаться неизменным: MySQL дополняет его нулевыми байтами до нужной длины.

MySQL сортирует столбцы `BLOB` и `TEXT` иначе, чем столбцы других типов: вместо сортировки строки по всей ее длине она сортирует только по первым `max_sort_length` байтам каждого столбца. Если нужна сортировка только по нескольким первым символам, вы можете либо уменьшить значение серверной переменной `max_sort_length`, либо использовать конструкцию `ORDER BY SUBSTRING(`*столбец*, *длина*`)`.

MySQL не может индексировать данные этих типов по полной длине и не может применять для сортировки индексы (в следующей главе мы рассмотрим этот вопрос подробнее).

Временные таблицы на диске и сортировка файлов

Поскольку подсистема хранения Memory не поддерживает типы `BLOB` и `TEXT`, для запросов, в которых используются столбцы такого типа и которым нужна неявная временная таблица, придется задействовать временные таблицы MyISAM на диске, даже если речь идет всего лишь о нескольких строках. Подсистема хранения Memory в пакете Percona Server поддерживает типы `BLOB` и `TEXT`, но во время написания книги она все еще не предотвращала использования таблиц на диске.

Это может потребовать серьезных затрат. Даже если вы настроите в MySQL хранение временных таблиц на диске в памяти, все равно потребуется много затратных вызовов операционной системы.

Лучше всего не использовать типы `BLOB` и `TEXT`, если можно без них обойтись. Если же избежать этого не удается, можно задействовать конструкцию `SUBSTRING(`*столбец, длина*`)` везде, где применяется тип `BLOB` (в том числе в разделе `ORDER BY`), для преобразования значений в символьные строки, которые уже могут храниться во временных таблицах в памяти. Только убедитесь, что выбираете достаточно короткие подстроки, чтобы временная таблица не вырастала до объемов, превышающих значения переменных `max_heap_table_size` или `tmp_table_size`, иначе MySQL преобразует ее в таблицу MyISAM на диске.

Самая неприятная ситуация с размером относится также к сортировке значений, поэтому этот трюк может помочь при обеих проблемах: создании больших временных таблиц и сортировке файлов и их создании на диске.

Приведем пример. Предположим, у вас есть таблица с 10 миллионами строк, которая занимает пару гигабайт на диске. В таблице есть столбец `VARCHAR(1000)` с кодировкой UTF8. Эта кодировка может использовать до 3 байт на один символ, что в худшем случае составит 3000 байт. Если вы укажете этот столбец в разделе `ORDER BY`, запрос ко всей таблице может потребовать более 30 Гбайт временного пространства только для сортировки файлов!

Если столбец `Extra` в результате применения команды `EXPLAIN` содержит слова `Using temporary`, значит, для запроса использована неявная временная таблица.

Использование типа ENUM вместо строкового типа

Иногда взамен обычных строковых типов можно задействовать тип `ENUM`. Столбец `ENUM` может хранить предопределенный набор различных строковых значений. MySQL сохраняет их очень компактно, упаковывая в 1 или 2 байта в зависимости от количества значений в списке. Она хранит каждое значение внутри себя как целое число, отражающее позицию его значения в списке значений поля, и сохраняет справочную таблицу, которая определяет соответствие между числом и строкой в файле `.frm`.

Приведем пример:

```
mysql> CREATE TABLE enum_test(
    ->   e ENUM('fish', 'apple', 'dog') NOT NULL
    -> );
mysql> INSERT INTO enum_test(e) VALUES('fish'), ('dog'), ('apple');
```

Во всех трех строках таблицы в действительности хранятся целые числа, а не строки. Вы можете увидеть двойственную природу значений, если извлечете их в числовом виде:

```
mysql> SELECT e + 0 FROM enum_test;
+-------+
| e + 0 |
+-------+
|     1 |
|     3 |
|     2 |
+-------+
```

Эта двойственность может вызвать путаницу, если вы определите числа в качестве констант `ENUM`, например `ENUM('1', '2', '3')`. Мы не рекомендуем так делать.

Другим сюрпризом является то, что поля типа `ENUM` сортируются по внутренним целочисленным значениям, а не по самим строкам:

```
mysql> SELECT e FROM enum_test ORDER BY e;
+-------+
| e     |
+-------+
| fish  |
| apple |
| dog   |
+-------+
```

Проблему можно решить, определив значения для столбца `ENUM` в желаемом порядке сортировки. Можно также использовать функцию `FIELD()`, чтобы явно задать порядок сортировки в запросе, но это не позволит MySQL применить для сортировки индекс:

```
mysql> SELECT e FROM enum_test ORDER BY FIELD(e, 'apple', 'dog', 'fish');
+-------+
| e     |
+-------+
| apple |
| dog   |
| fish  |
+-------+
```

Если бы мы определили значения в алфавитном порядке, нам не пришлось бы это делать.

Главным недостатком столбцов типа ENUM является то, что список строк фиксирован, а для их добавления или удаления необходимо использовать команду ALTER TABLE. Таким образом, возможно, не слишком хорошая идея — применить тип ENUM для представления строк, если предполагается изменить список возможных значений в будущем, за исключением случая, когда допустимо добавлять элементы в конец списка, что можно сделать без полного перестроения таблицы в MySQL 5.1.

Поскольку MySQL сохраняет каждое значение как целое число и вынуждена просматривать таблицы соответствий для преобразования их в строковое представление, то со столбцами типа ENUM связаны некоторые издержки. Обычно это компенсируется их малым размером, но так происходит не всегда. В частности, соединение столбца типа CHAR или VARCHAR со столбцом типа ENUM может оказаться медленнее, чем соединение с другим столбцом типа CHAR или VARCHAR.

Чтобы проиллюстрировать данное утверждение, мы провели эталонное тестирование того, как быстро MySQL выполняет такое соединение с таблицей, в одном из наших приложений. В таблице определили довольно большой первичный ключ:

```
CREATE TABLE webservicecalls (
    day date NOT NULL,
    account smallint NOT NULL,
    service varchar(10) NOT NULL,
    method varchar(50) NOT NULL,
    calls int NOT NULL,
    items int NOT NULL,
    time float NOT NULL,
    cost decimal(9,5) NOT NULL,
    updated datetime,
    PRIMARY KEY  (day, account, service, method)
) ENGINE=InnoDB;
```

Таблица содержит около 110 000 строк, ее объем порядка 10 Мбайт, поэтому она целиком помещается в памяти. Столбец service содержит пять различных значений со средней длиной четыре символа, а столбец method — 71 значение со средней длиной 20 символов.

Мы сделали копию этой таблицы и преобразовали столбцы service и method к типу ENUM следующим образом:

```
CREATE TABLE webservicecalls_enum (
    ... опущено ...
    service ENUM(...значения опущены...) NOT NULL,
    method ENUM(...значения опущены...) NOT NULL,
   ... опущено ...
 ) ENGINE=InnoDB;
```

Затем измерили производительность соединения таблиц по столбцам первичного ключа. Для этого использовали такой запрос:

```
mysql> SELECT SQL_NO_CACHE COUNT(*)
    -> FROM webservicecalls
    ->    JOIN webservicecalls USING(day, account, service, method);
```

Мы варьировали этот запрос, соединяя столбцы типа VARCHAR и ENUM в различных комбинациях. Результаты показаны в табл. 4.1.

Таблица 4.1. Скорость соединения столбцов типа VARCHAR и ENUM

Тест	Запросов в секунду
Соединение VARCHAR с VARCHAR	2,6
Соединение VARCHAR с ENUM	1,7
Соединение ENUM с VARCHAR	1,8
Соединение ENUM с ENUM	3,5

Соединение становится быстрее после преобразования всех столбцов к типу ENUM, но соединение столбцов типа ENUM со столбцами типа VARCHAR происходит медленнее. В данном случае кажется, что преобразование имеет смысл, если не планируется соединение этих столбцов со столбцами типа VARCHAR. Распространенная практика проектирования состоит в использовании справочных таблиц с целочисленными первичными ключами, чтобы избежать соединения по символьным значениям.

Однако можно получить пользу и от преобразования столбцов: команда SHOW TABLE STATUS в столбце Data_length показывает, что после преобразования двух столбцов к типу ENUM таблица стала приблизительно на треть меньше. В некоторых ситуациях это может принести выгоду даже в случае соединения столбцов типа ENUM со столбцами типа VARCHAR. Кроме того, сам первичный ключ после преобразования уменьшается приблизительно вдвое. Поскольку это таблица InnoDB, то если в ней есть и другие индексы, поэтому уменьшение размера первичного ключа приведет к уменьшению и их размера. Мы поясним это в следующей главе.

Типы Date и Time

MySQL содержит много различных типов для даты и времени, например YEAR и DATE. Минимальной единицей времени, которую может хранить MySQL, является 1 секунда. (У MariaDB минимальная единица временно́го типа — 1 микросекунда.) Однако *вычисления* с временными данными могут выполняться с точностью до 1 микросекунды, и мы покажем, как обойти ограничение хранения.

Большинство временных типов не имеют альтернатив, поэтому вопроса о том, какой из них лучше, не возникает. Нужно лишь решить, что делать, когда требуется сохранять и дату, и время. Для этих целей MySQL предлагает два очень похожих типа данных: DATETIME и TIMESTAMP. Для большинства приложений подходят оба, но в некоторых случаях один работает лучше, чем другой. Давайте рассмотрим их более подробно.

- DATETIME. Этот тип дает возможность хранить большой диапазон значений — с 1001 по 9999 год — с точностью до 1 секунды. Дата и время упаковываются в целое число в формате YYYYMMDDHHMMSS независимо от часового пояса. Для хранения требуется 8 байт.

По умолчанию MySQL показывает данные типа `DATETIME` в точно определенном допускающем сортировку формате: 2008-01-16 22:37:08. Это стандарт представления даты и времени ANSI.

- `TIMESTAMP`. Как следует из названия, в типе `TIMESTAMP` хранится количество секунд, прошедших после полуночи 1 января 1970 года по Гринвичу (GMT), — и как во временной метке UNIX. Для хранения данных типа `TIMESTAMP` используются только 4 байта, поэтому он позволяет представить значительно меньший диапазон дат, чем тип `DATETIME`: с 1970 года до некоторой даты в 2038 году. В MySQL имеются функции `FROM_UNIXTIME()` и `UNIX_TIMESTAMP()`, служащие для преобразования временной метки UNIX в дату и наоборот.

  В MySQL 4.1 и более новых версиях значения типа `TIMESTAMP` форматируются точно так же, как значения `DATETIME`, но в MySQL 4.0 и более старых версиях они отображаются без разделителей между частями. Это различие проявляется только при выводе — формат хранения значений `TIMESTAMP` одинаков во всех версиях MySQL.

  Отображаемое значение типа `TIMESTAMP` зависит также от часового пояса. Часовой пояс определен для сервера MySQL, операционной системы и клиентского соединения.

  Таким образом, если в поле типа `TIMESTAMP` хранится значение `0`, то для часового пояса Eastern Standard Time (EST), отличающегося от гринвичского времени на 5 часов, будет выведена строка `1969-12-31 19:00:00`. Стоит подчеркнуть эту разницу: если вы храните данные, относящиеся к нескольким часовым поясам, или получаете к ним доступ, поведение `TIMESTAMP` и `DATETIME` будет различаться. Первый сохраняет значения относительно используемого часового пояса, а последний — текстовое представление даты и времени.

  Кроме того, у типа `TIMESTAMP` есть специальные свойства, которых нет у типа `DATETIME`. По умолчанию, если вы не указали значение для столбца, MySQL вставляет в первый столбец типа `TIMESTAMP` текущее время[1]. Кроме того, по умолчанию MySQL изменяет значение первого столбца типа `TIMESTAMP` при обновлении строки, если ему явно не присвоено значение в команде `UPDATE`. Вы можете настроить поведение при вставке и обновлении для каждого столбца типа `TIMESTAMP`. Наконец, столбцы типа `TIMESTAMP` по умолчанию создаются в режиме `NOT NULL`, в отличие от остальных типов данных.

В общем случае мы советуем пользоваться типом `TIMESTAMP`, если это возможно, поскольку с точки зрения занимаемого на диске места он намного эффективнее, чем `DATETIME`. Иногда временные метки UNIX сохраняют в виде целых чисел, но обычно это не дает никаких преимуществ. Целочисленное представление часто неудобно, поэтому мы не рекомендуем его применять.

---

[1] Правила поведения типа TIMESTAMP сложны и неодинаковы в различных версиях MySQL, поэтому нужно убедиться, что он ведет себя так, как хочется вам. Обычно имеет смысл проверить вывод команды SHOW CREATE TABLE после внесения изменений в столбцы TIMESTAMP.

Но что, если вам нужно сохранить значение даты и времени с точностью выше секунды? Сейчас в MySQL для этих целей нет соответствующего типа данных, но вы можете создать собственный формат хранения: применить тип `BIGINT` и сохранить значение как временную метку в микросекундах либо воспользоваться типом `DOUBLE` и указать дробную часть секунды после запятой. Оба подхода вполне работоспособны. Или вы можете воспользоваться MariaDB вместо MySQL.

Битовые типы данных

В MySQL есть несколько типов данных, использующих для компактного хранения значений отдельные биты. Все эти типы с технической точки зрения являются строковыми вне зависимости от формата хранения и обработки.

- `BIT`. До версии MySQL 5.0 ключевое слово `BIT` было просто синонимом `TINYINT`. Но начиная с версии MySQL 5.0 это совершенно иной тип данных с особыми характеристиками. Обсудим его поведение.

  Вы можете использовать столбец типа `BIT` для хранения одного или нескольких значений `true/false` в одном столбце. `BIT(1)` определяет поле, содержащее 1 бит, `BIT(2)` хранит 2 бита и т. д. Максимальная длина столбца типа `BIT` равна 64 битам.

  Поведение типа `BIT` зависит от подсистемы хранения. MyISAM для хранения упаковывает битовые столбцы, поэтому для хранения 17 отдельных столбцов типа `BIT` требуется только 17 бит (предполагается, что ни в одном из столбцов не разрешено значение `NULL`). При вычислении размера места для хранения MyISAM округлит это число до 3 байт. Другие подсистемы, например Memory и InnoDB, хранят каждый столбец как наименьший целочисленный тип, достаточно большой для размещения всех битов, поэтому сэкономить пространство не получится.

  MySQL рассматривает `BIT` как строковый, а не числовой тип. Когда вы извлекаете значение типа `BIT(1)`, результатом является строка, но ее содержимое представляет собой двоичное значение 0 или 1, а не значение 0 или 1 в кодировке ASCII. Однако, если вы извлечете значение в числовом виде, результатом будет число, в которое преобразуется битовая строка. Учитывайте это, когда понадобится сравнить результат с другим значением. Например, если вы сохраните значение `b'00111001'`, являющееся двоичным эквивалентом числа 57, в столбце типа `BIT(8)`, а затем извлечете его, то получите строку, содержащую код символов 57. Этот код в кодировке ASCII соответствует цифре 9. Но в числовом виде вы получите значение `57`:

```
mysql> CREATE TABLE bittest(a bit(8));
mysql> INSERT INTO bittest VALUES(b'00111001');
mysql> SELECT a, a + 0 FROM bittest;
+------+-------+
| a    | a + 0 |
+------+-------+
| 9    |    57 |
+------+-------+
```

Такое поведение может сбивать с толку, так что рекомендуем с осторожностью использовать тип `BIT`. В большинстве приложений лучше вообще его избегать.

Если возникла необходимость сохранять значение `true/false` в одном бите, можно также создать столбец типа `CHAR(0)` с возможностью хранения `NULL`. Такой столбец может хранить как отсутствие значения (`NULL`), так и значение нулевой длины (пустая строка).

- `SET`. Если нужно хранить множество значений `true/false`, попробуйте объединить несколько столбцов в один естественный для MySQL столбец типа `SET`. В MySQL его внутренним представлением является упакованный битовый вектор. Он эффективно использует пространство, а в MySQL есть такие функции, как `FIND_IN_SET()` и `FIELD()`, которые облегчают запросы. Основным недостатком являются затраты на изменение определения столбца — эта процедура требует команды `ALTER TABLE`, которая для больших таблиц обходится очень дорого (но далее в этой главе мы рассмотрим обходной путь). Кроме того, в общем случае при поиске в столбцах типа `SET` нельзя применять индексы.
- *Побитовые операции над целочисленными столбцами.* Альтернативой типу `SET` является использование целого числа как упакованного набора битов. Например, вы можете поместить 8 бит в тип `TINYINT` и выполнять с ним побитовые операции. Для упрощения работы можно определить именованные константы для каждого бита в коде приложения.

  Главным преимуществом такого подхода по сравнению с использованием типа `SET` является то, что вы можете изменить перечисление, которое представляет поле без команды `ALTER TABLE`. Недостаток же заключается в том, что запросы становятся труднее для написания и понимания (что означает, если бит 5 установлен?). Одни люди хорошо ориентируются в побитовых операциях, а другие — нет, поэтому использование описанного метода — дело вкуса.

Примером применения упакованных битов является список управления доступом (access control list, ACL), в котором хранятся разрешения. Каждый бит или элемент `SET` представляет значение, например `CAN_READ`, `CAN_WRITE` или `CAN_DELETE`. Если вы используете столбец типа `SET`, то позволяете MySQL сохранять в определении столбца соответствие между битами и значениями; если же применяется целочисленный столбец, то такое соответствие устанавливается в коде приложения. Приведем примеры запросов со столбцом типа `SET`:

```
mysql> CREATE TABLE acl (
    ->    perms SET('CAN_READ', 'CAN_WRITE', 'CAN_DELETE') NOT NULL
    -> );
mysql> INSERT INTO acl(perms) VALUES ('CAN_READ,CAN_DELETE');
mysql> SELECT perms FROM acl WHERE FIND_IN_SET('CAN_READ', perms);
+--------------------+
| perms              |
+--------------------+
| CAN_READ,CAN_DELETE |
+--------------------+
```

При использовании целочисленного столбца вы могли бы написать этот пример следующим образом:

```
mysql> SET @CAN_READ   := 1 << 0,
    ->     @CAN_WRITE  := 1 << 1,
    ->     @CAN_DELETE := 1 << 2;
mysql> CREATE TABLE acl (
    ->    perms TINYINT UNSIGNED NOT NULL DEFAULT 0
    -> );
mysql> INSERT INTO acl(perms) VALUES(@CAN_READ + @CAN_DELETE);
mysql> SELECT perms FROM acl WHERE perms & @CAN_READ;
+-------+
| perms |
+-------+
|     5 |
+-------+
```

Для определения столбцов мы использовали переменные, но вы можете применять в своем коде и константы.

Выбор идентификаторов

Выбор типа данных для столбца идентификатора очень важен. Велика вероятность того, что этот столбец будет сравниваться с другими значениями (например, в соединениях) и использоваться для поиска чаще, чем другие столбцы. Кроме того, вы, возможно, станете применять идентификаторы в качестве внешних ключей в других таблицах, поэтому выбор типа столбца идентификатора, скорее всего, определит и типы столбцов в связанных таблицах. (Как мы уже показали, целесообразно использовать в связанных таблицах одинаковые типы данных, поскольку они, скорее всего, будут задействованы для соединения.)

При выборе типа данных для столбца идентификатора следует учитывать не только тип хранения, но и то, как MySQL выполняет вычисления и сравнения с этим типом. Например, MySQL хранит типы `ENUM` и `SET` как целые числа, но преобразует их в строки при сравнении в строковом контексте.

Сделав выбор, убедитесь, что вы используете один и тот же тип во всех связанных таблицах. Типы должны совпадать в точности, включая такие свойства, как `UNSIGNED`[1]. Смешение различных типов данных может вызвать проблемы с производительностью, и даже если этого не произойдет, неявные преобразования типов в процессе сравнения могут привести к возникновению труднообнаруживаемых ошибок.

[1] При использовании подсистемы хранения InnoDB в принципе невозможно создать внешний ключ, если типы данных в точности не совпадают. Появляющееся сообщение об ошибке ERROR 1005 (HY000): Can't create table может сбить с толку (в определенном контексте), а вопросы на эту тему часто появляются в списках рассылок MySQL (как ни странно, разрешается создавать внешние ключи между столбцами VARCHAR разной длины).

Они могут проявиться значительно позже, когда вы уже забудете, что сравниваете данные разных типов.

Выбирайте самый маленький размер поля, способный вместить требуемый диапазон значений, а при необходимости оставляйте место для увеличения в дальнейшем. Например, если вы храните названия штатов США в столбце `state_id`, в нем не будет тысяч или миллионов значений, поэтому не используйте тип `INT`. Вполне достаточно типа `TINYINT`, который на 3 байта короче. Если вы используете это значение как внешний ключ в других таблицах, 3 байта могут стать существенными. Дарим несколько советов.

- *Целочисленные типы.* Обычно хорошо подходят для идентификаторов, поскольку работают быстро и допускают `AUTO_INCREMENT`.
- `ENUM` и `SET`. Типы `ENUM` и `SET` обычно не подходят для идентификаторов, хотя могут быть хороши для статических таблиц определений, содержащих значения состояния или типа. Столбцы `ENUM` и `SET` подходят для хранения такой информации, как состояние заказа, вид продукта или пол человека.

  Например, если поле `ENUM` используется для определения вида продукта, вам может потребоваться справочная таблица с первичным ключом по идентичному полю `ENUM` (можете включить в справочную таблицу столбцы с текстом описания, создать глоссарий или добавить комментарии в элементах раскрывающегося меню на сайте). В таком случае можно использовать тип `ENUM` для идентификатора, но в целом лучше этого избегать.
- *Строковые типы.* По возможности избегайте строковых типов для идентификаторов, поскольку они занимают много места и обычно обрабатываются медленнее целочисленных. Особенно осторожными следует быть при использовании строковых идентификаторов в таблицах MyISAM. Подсистема хранения MyISAM по умолчанию использует упакованные индексы для строк, что замедляет поиск. Во время тестов мы обнаружили, что в процессе работы с упакованными индексами MyISAM производительность падает чуть ли не в шесть раз.

  Следует также быть очень внимательными при работе со случайными строками, например сгенерированными функциями `MD5()`, `SHA1()` или `UUID()`. Случайные значения, созданные с их помощью, распределяются случайным образом в большом пространстве, что может замедлить работу команды `INSERT` и некоторых типов запросов `SELECT`[1].

  - Они замедляют запросы `INSERT`, поскольку вставленное значение должно быть помещено в случайное место в индексах. Это приводит к разделению страниц, появлению возможности произвольного доступа к диску и фрагментации кластерного индекса в подсистемах хранения, которые его поддерживают. Более подробно мы рассмотрим это в следующей главе.

[1] В то же время в больших таблицах, в которые одновременно записывают данные множество клиентов, такие псевдослучайные значения могут помочь избежать «горячих» мест.

- Они замедляют запросы `SELECT`, так как логически соседние строки оказываются разбросанными по всему диску и памяти.
- Случайные значения ухудшают работу кэша при всех типах запросов, поскольку нарушают принцип локальности ссылок, лежащий в основе его работы. Если весь набор данных одинаково «горячий», то нет смысла в хранении какой-то части информации кэшированной в памяти, а если рабочие данные не помещаются в памяти, то часто будут возникать сбросы из кэша.

Если вы решили сохранять значения `UUID`, то нужно удалить тире или, что еще лучше, преобразовать значения `UUID` в 16-байтные числа с помощью функции `UNHEX()` и сохранить их в столбце типа `BINARY(16)`. Вы можете извлекать значения в шестнадцатеричном формате с применением функции `HEX()`.

Характеристики значений, сгенерированных с помощью функции `UUID()`, отличаются от сгенерированных криптографическими хеш-функциями, такими как `SHA1()`: данные `UUID` распределены неравномерно и являются в некоторой степени последовательными. Однако они не настолько хороши, как монотонно увеличивающиеся целые числа.

Остерегайтесь автоматически сгенерированных схем

Мы описали наиболее важные особенности типов данных (одни больше влияют на производительность, другие — меньше), но еще не говорили о недостатках автоматически сгенерированных схем.

Плохо написанные программы миграции схем и программы, автоматически генерирующие схемы, могут вызывать серьезные проблемы с производительностью. Некоторые программы создают большие поля `VARCHAR` для *всех* типов данных или используют различные типы данных для столбцов, которые будут сравниваться при соединениях. Тщательно изучите схему, если она была сгенерирована автоматически.

Системы объектно-реляционного отображения (object-relational mapping, ORM) (и фреймворки, которые они используют) — еще один кошмар для желающих достичь высокой производительности. Некоторые из этих систем позволяют хранить любой тип данных в произвольном хранилище, а это обычно означает, что они не могут использовать сильные стороны конкретного хранилища. Иногда они записывают каждое свойство объекта в отдельную строку с использованием хронологического контроля версий, в результате чего возникает сразу несколько версий каждого свойства!

Возможно, такой подход привлекателен для разработчиков, поскольку позволяет им применять объектно-ориентированный стиль, не задумываясь о том, как данные хранятся. Однако приложения, скрывающие сложность от разработчиков, обычно плохо масштабируются. Мы советуем хорошо подумать, прежде чем променять производительность на удобство разработки. Всегда тестируйте на реалистичных наборах данных, чтобы проблемы с производительностью не были выявлены слишком поздно.

Специальные типы данных

Некоторые типы данных не соответствуют встроенным типам. Одним из них является временна́я метка с точностью более 1 секунды. Несколько способов сохранения подобных данных мы уже показали в этой главе.

Другой пример — IPv4-адреса. Для их хранения часто используют столбцы типа `VARCHAR(15)`. Однако на деле IP-адрес является 32-битным беззнаковым целым числом, а не строкой. Запись с точками, разделяющими байты, предназначена только для того, чтобы человеку было удобнее его воспринимать. Лучше хранить IP-адреса как беззнаковые целые числа. У MySQL есть функции `INET_ATON()` и `INET_NTOA()` для преобразования между двумя представлениями.

Подводные камни проектирования схемы в MySQL

Несмотря на то что существуют как хорошие, так и плохие принципы проектирования, есть и проблемы, возникающие из-за особенностей реализации MySQL, что означает возможность возникновения характерных только для MySQL ошибок. В данном разделе обсуждаются проблемы, которые мы наблюдали в схемах проектирования в MySQL. Этот материал может помочь вам избежать ошибок и выбрать альтернативы, которые лучше работают в MySQL.

- *Слишком много столбцов.* Во время работы MySQL строки копируются между сервером и подсистемой хранения данных в формате строкового буфера, затем сервер преобразует буфер в столбцы. Однако преобразование из строкового буфера в структуру строковых данных может оказаться весьма затратным. Формат фиксированной строки MyISAM точно соответствует формату строки сервера, поэтому преобразование не требуется. Однако переменный формат строк MyISAM и формат строк InnoDB всегда требуют преобразования. Затраты на него зависят от количества столбцов. Мы обнаружили, что эта процедура может быть очень затратной при высоком уровне использования процессора и огромных таблицах (сотни столбцов), даже если фактически использовалось лишь несколько столбцов. Если вы планируете задействовать сотни столбцов, имейте в виду, что характеристики производительности сервера будут разными.
- *Слишком много соединений.* Паттерн проектирования «сущность — атрибут — значение» (entity — attribute — value, EAV) является классическим примером неудачного паттерна проектирования, который особенно плохо работает в MySQL. В MySQL существует ограничение на 61 таблицу для каждого соединения, а для баз данных с паттерном EAV требуется множество самосоединений. Мы неоднократно видели, как базы данных, использующие паттерн EAV, в конечном счете превышают этот предел. Однако даже при гораздо меньшем чем 61 количестве соединений затраты на планирование и оптимизацию запроса могут вызывать проблемы MySQL. Эмпирическое правило гласит: если запросы должны выполняться быстро в условиях высокого параллелизма, сделайте десятки таблиц для каждого запроса.

- *Всемогущий тип ENUM.* Остерегайтесь чрезмерного использования типа ENUM. Приведем пример из реальной жизни:

```
CREATE TABLE ... (
  country enum('','0','1','2',...,'31')
```

Схема была обильно сдобрена этим паттерном. Скорее всего, такое проектное решение было бы сомнительным в любой базе данных с перечисляемым типом значений. В действительности здесь должно использоваться целочисленное значение, являющееся внешним ключом таблицы словаря или справочника. Но в MySQL возникают дополнительные проблемы. Новую страну в этот список можно добавить только с помощью команды ALTER TABLE, которая является блокирующей операцией в MySQL 5.0 и более ранних версиях и остается таковой в версии 5.1 и более поздних, за исключением случая, когда добавление происходит в конец списка. (Далее покажем пару трюков для решения этой проблемы, но это всего лишь трюки.)

- *Замаскированный тип ENUM.* Тип ENUM позволяет столбцу иметь одно значение из набора определенных значений. SET позволяет столбцу иметь одно или несколько значений из набора определенных значений. Иногда их легко перепутать. Приведем пример:

```
CREATE TABLE ...(
  is_default set('Y','N') NOT NULL default 'N'
```

Здесь почти наверняка вместо SET должен быть ENUM, конечно, при условии, что значение не может быть одновременно истинным и ложным.

- *NULL изобрели не здесь.* Ранее мы рекомендовали не использовать NULL и, действительно, советуем по возможности применять альтернативные варианты. Даже если нужно хранить факт «нет значения» в таблице, вы можете обойтись без NULL. Как вариант можете использовать нуль, специальное значение или пустую строку.

Однако в крайнем случае можно использовать и NULL. Не бойтесь этого, если нужно представить неизвестное значение. Иногда лучше указать NULL, чем магическую константу. Выбор одного значения из ограниченного набора типов, например -1 для представления неизвестного целого числа, может очень усложнить ваш код, внести в него ошибки и просто добавить беспорядка. NULL не всегда легко обрабатывается, но часто он лучше альтернативных вариантов.

Приведем неоднократно виденный нами пример:

```
CREATE TABLE ... (
  dt DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00'
```

Это фиктивное значение (все нули) способно вызвать множество проблем. (Вы можете настроить конфигурационную переменную SQL_MODE, чтобы запретить бессмысленные даты, что является особенно удачным приемом для нового приложения, которое еще не успело наполнить базу плохими данными.)

Одним глазком заглянем в другой раздел и отметим, что MySQL все-таки индексирует NULL, в отличие от Oracle, который включает в индексы только значения.

Нормализация и денормализация

Как правило, существует множество способов представить имеющиеся данные: от полной нормализации до полной денормализации со всеми промежуточными вариантами. В нормализованной базе данных каждый факт представлен один и только один раз. В денормализованной базе данных, наоборот, информация дублируется или хранится в нескольких местах.

Если вы не знакомы с нормализацией, вам стоит изучить ее. На эту тему существует много хороших книг и веб-ресурсов, здесь же мы дадим только краткое введение в те темы, которые вы должны знать, чтобы понять эту главу. Начнем с классического примера с сотрудниками (employee), подразделениями (department) и руководителями подразделений (department head):

EMPLOYEE	DEPARTMENT	HEAD
Jones	Accounting	Jones
Smith	Engineering	Smith
Brown	Accounting	Jones
Green	Engineering	Smith

Проблема с этой схемой состоит в том, что при модификации данных могут возникнуть противоречия. Предположим, Браун стал начальником бухгалтерии. Чтобы отразить это изменение, нужно обновить множество строк, а это трудоемко и чревато возникновением ошибок. Если в строке `Jones` указан начальник отдела, отличный от того, кто указан в строке `Brown`, невозможно выяснить, где правда. Это как в поговорке «Человек, у которого двое часов, никогда не знает точного времени». Кроме того, мы не можем сохранить информацию о подразделении без сотрудников — если удалим всех сотрудников бухгалтерии, то потеряем информацию о самом подразделении. Чтобы избежать этого, нужно нормализовать таблицу, разделив списки сотрудников и подразделений. Результатом этого процесса будет появление двух следующих таблиц — для сотрудников:

EMPLOYEE_NAME	DEPARTMENT
Jones	Accounting
Smith	Engineering
Brown	Accounting
Green	Engineering

и подразделений:

DEPARTMENT	HEAD
Accounting	Jones
Engineering	Smith

Теперь эти таблицы приведены ко второй нормальной форме, вполне подходящей для многих задач. Однако вторая нормальная форма — это только одна из множества возможных нормальных форм.

Здесь мы использовали фамилию как первичный ключ, только чтобы проиллюстрировать пример, поскольку это естественный идентификатор данных. Однако на практике мы бы не стали так делать. Нет гарантии, что фамилии не повторяются, а кроме того, нецелесообразно использовать в качестве первичных ключей длинные строки.

Достоинства и недостатки нормализованной схемы

Людям, которые просят помочь им справиться с проблемами производительности, часто советуют провести нормализацию, особенно если рабочая нагрузка характеризуется большим количеством операций записи. Нередко этот совет оказывается верным, особенно в следующих ситуациях.

- Нормализованные таблицы обычно обновляются быстрее, чем ненормализованные.
- Когда данные хорошо нормализованы, они либо редко дублируются, либо не дублируются совсем. Так что изменять приходится меньше данных.
- Нормализованные таблицы обычно меньше по размеру, чем ненормализованные, поэтому лучше помещаются в памяти и их производительность выше.
- Отсутствие избыточных данных означает, что для извлечения списков значений реже требуется использовать в запросах ключевое слово `DISTINCT` или раздел `GROUP BY`. Рассмотрим предыдущий пример: из денормализованной схемы невозможно получить отдельный список подразделений без `DISTINCT` или `GROUP BY`, но если `DEPARTMENT` является отдельной таблицей, запрос будет элементарным.

Недостатки нормализованной схемы обычно проявляются при извлечении данных. Любой нестандартный запрос к хорошо нормализованной схеме, скорее всего, потребует по крайней мере одного, а то и нескольких соединений. Это не только затратно, но и делает невозможными некоторые стратегии индексирования. Например, при нормализации в разных таблицах могут оказаться столбцы, которые целесообразнее было бы иметь в одном индексе.

Достоинства и недостатки денормализованной схемы

Денормализованная схема работает хорошо, поскольку все данные находятся в одной и той же таблице, что позволяет избежать соединений.

Если вам не нужно соединять таблицы, то худшим вариантом для большинства запросов, даже тех, которые не используют индексы, является сканирование таблицы.

Оно может выполняться быстрее соединения, если данные не помещаются в память, поскольку в этом случае удастся избежать операций ввода/вывода с произвольным доступом.

Кроме того, единая таблица допускает применение более эффективных стратегий индексирования. Предположим, у вас есть сайт, где посетители оставляют свои сообщения, при этом некоторые пользователи являются привилегированными. Допустим, нужно просмотреть десять последних сообщений от привилегированных пользователей. Если вы нормализовали схему и индексировали даты публикации сообщений, запрос может выглядеть примерно так:

```
mysql> SELECT message_text, user_name
    -> FROM message
    ->    INNER JOIN user ON message.user_id=user.id
    -> WHERE user.account_type='premium
    -> ORDER BY message.published DESC LIMIT 10;
```

Для эффективного выполнения этого запроса MySQL потребуется просканировать индекс `published` в таблице `message`. Для каждой найденной строки в таблице `user` придется проверять, является ли пользователь привилегированным. Такая обработка будет неэффективной, если доля привилегированных пользователей невелика.

Другой возможный план запроса заключается в том, чтобы начать с таблицы `user`, выбрать всех привилегированных пользователей, получить для них все сообщения и выполнить файловую сортировку. Это, по-видимому, будет еще хуже.

Проблематичным является соединение, которое не позволяет сортировать и фильтровать одновременно с помощью единственного индекса. Если вы денормализуете данные, объединив таблицы, и добавите индекс `(account_type, published)`, то сможете написать запрос без соединения. Это будет очень эффективно:

```
mysql> SELECT message_text,user_name
    -> FROM user_messages
    -> WHERE account_type='premium'
    -> ORDER BY published DESC
    -> LIMIT 10;
```

Сочетание нормализации и денормализации

Как, зная о преимуществах и недостатках нормализованных и денормализованных схем, выбрать лучший вариант?

Истина заключается в том, что полностью нормализованные и денормализованные схемы редко имеют что-то общее с реальным миром. На практике часто приходится сочетать оба подхода, применяя частично нормализованные схемы, кэшированные таблицы и другие приемы.

Самым общим способом денормализации данных является дублирование или кэширование отдельных столбцов из одной таблицы в другую. В MySQL 5.0 и более новых версиях вы можете использовать триггеры для обновления кэшированных значений, что облегчает реализацию задачи.

Например, в нашем примере сайта вместо выполнения полной денормализации вы можете хранить значение `account_type` как в таблице `user`, так и в таблице `message`. Это поможет избежать проблем при вставке и удалении данных, которые возникли бы при полной денормализации, поскольку вы никогда не потеряете информацию о пользователе, даже если сообщений нет. Это ненамного увеличит размеры таблицы `user_message`, но позволит эффективно выбирать данные.

Однако теперь обновление типа учетной записи пользователя становится более затратным, поскольку вам придется изменять ее в обеих таблицах. Чтобы понять, действительно ли это станет проблемой, нужно оценить, как часто будут выполняться такие изменения и сколько времени они будут занимать, а затем сравнить с частотой выполнения запросов `SELECT`.

Еще одной причиной переноса некоторых данных из родительской таблицы в дочернюю может стать сортировка. Например, в нормализованной схеме сортировка сообщений по имени автора будет чрезвычайно затратной операцией. Однако, если вы кэшируете столбец `author_name` в таблице `message` и проиндексируете ее, сортировка может пройти очень эффективно.

Может также оказаться полезным кэшировать производные значения. Если вам нужно показывать, сколько сообщений оставил каждый пользователь (как это делается на многих форумах), то можно либо запустить затратный подзапрос, который будет считать сообщения при каждом отображении, либо добавить в таблицу `user` столбец `num_messages`, который станет обновляться каждый раз, когда пользователь оставит новое сообщение.

Кэшированные и сводные таблицы

В некоторых случаях наилучший способ увеличения производительности состоит в сохранении избыточных данных в той же таблице, где находятся исходные данные. Однако иногда требуется построить отдельную сводную или кэшированную таблицу, специально настроенную под ваши потребности по извлечению данных. Этот метод хорошо работает, если вы готовы смириться с некоторым устареванием информации, но иногда у вас просто нет другого выхода (например, когда нужно избежать сложных и затратных обновлений в режиме реального времени).

Термины «кэшированная таблица» и «сводная таблица» не являются общепринятыми. Мы используем понятие «кэшированная таблица» для обозначения таблиц, содержащих данные, которые можно легко, хотя и более медленно, извлечь из схемы (то есть логически избыточные данные). Под сводными таблицами мы подразумеваем таблицы, в которых хранятся агрегированные данные из запросов с фразой `GROUP BY` (то есть данные, не являющиеся логически избыточными). Сводные таблицы иногда называют *свернутыми*, поскольку данные в них свернуты.

Продолжая рассматривать пример с сайтом, предположим, что требуется подсчитать количество сообщений, оставленных за последние 24 часа. На загруженном сайте поддерживать точный счетчик в режиме реального времени было бы невозможно.

Вместо этого вы можете каждый час генерировать сводную таблицу. Часто для этого хватает единственного запроса, и это эффективнее, чем поддерживать работающий в реальном времени счетчик. Недостаток же заключается в том, что полученные результаты не будут стопроцентно точными.

Если необходим точный подсчет сообщений, размещенных за предыдущие 24 часа (без устаревания данных), есть и другой вариант. Начнем с почасовой сводной таблицы. Затем вычислим точное количество сообщений, оставленных за данный 24-часовой период, суммируя число сообщений за 23 целых часа этого периода, неполный час в начале периода и неполный час в его конце. Предположим, что сводная таблица называется `msg_per_hr` и определена следующим образом:

```
CREATE TABLE msg_per_hr (
    hr DATETIME NOT NULL,
    cnt INT UNSIGNED NOT NULL,
    PRIMARY KEY(hr)
);
```

Можно получить количество сообщений, оставленных за предыдущие 24 часа, сложив результаты следующих трех запросов. Мы используем `LEFT(NOW(), 14)` для округления текущей даты и времени до ближайшего часа:

```
mysql> SELECT SUM(cnt) FROM msg_per_hr
    -> WHERE hr BETWEEN
    ->    CONCAT(LEFT(NOW(), 14), '00:00') - INTERVAL 23 HOUR
    ->    AND CONCAT(LEFT(NOW(), 14), '00:00') - INTERVAL 1 HOUR;
mysql> SELECT COUNT(*) FROM message
    -> WHERE posted >= NOW() - INTERVAL 24 HOUR
    ->    AND posted < CONCAT(LEFT(NOW(), 14), '00:00') - INTERVAL 23 HOUR;
mysql> SELECT COUNT(*) FROM message
    -> WHERE posted >= CONCAT(LEFT(NOW(), 14), '00:00');
```

Оба подхода — неточный подсчет или точное вычисление с запросами по небольшим диапазонам — эффективнее, чем подсчет всех строк в таблице `message`. Это основная причина создания сводных таблиц. Вычисление этой статистики в режиме реального времени чрезвычайно затратно, поскольку требуется либо просматривать большой объем данных, либо использовать запросы, эффективные только при наличии специальных индексов, которые вы не хотите добавлять из-за их влияния на обновления строк. Типичный пример таких операций — вычисление наиболее активных пользователей или наиболее часто используемых тегов.

В свою очередь, кэшированные таблицы полезны для оптимизации поиска и извлечения данных. Такие запросы часто требуют специальной структуры таблиц и индексов, отличной от той, которую обычно используют для обработки транзакций в реальном времени (online transaction processing, OLTP).

Например, возможно, вам требуется много комбинаций индексов для ускорения различных типов запросов. Эти конфликтующие требования иногда приводят к необходимости создать кэшированную таблицу, содержащую лишь некоторые столбцы главной таблицы. Как вариант, для кэшированной таблицы можно использовать другую подсистему хранения. Например, если главная таблица использует InnoDB, то, применив для кэшированной таблицы MyISAM, вы добьетесь

уменьшения размера индекса и возможности выполнять запросы с полнотекстовым поиском. Вероятно, в отдельных случаях имеет смысл полностью вынести таблицу из MySQL и поместить ее в специализированную систему, которая может более эффективно выполнять поиск, например в поисковые системы Lucene или Sphinx.

При использовании кэшированных и сводных таблиц вам придется решать, поддерживать их в режиме реального времени или периодически перестраивать. Выбор зависит от приложения, но периодическое перестроение не только экономит ресурсы, но и может дать более эффективную таблицу без фрагментации и с полностью отсортированными индексами.

При перестроении сводных и кэшированных таблиц часто требуется, чтобы во время этой операции хранящиеся в них данные оставались доступными. Этого можно добиться, используя теневую копию таблицы, то есть таблицу, созданную «за кулисами» основной. Закончив ее построение, вы можете поменять таблицы местами, просто переименовав их. Например, если вам нужно перестроить таблицу `my_summary`, можете создать таблицу `my_summary_new`, заполнить ее данными и поменять местами с реальной таблицей:

```
mysql> DROP TABLE IF EXISTS my_summary_new, my_summary_old;
mysql> CREATE TABLE my_summary_new LIKE my_summary;
-- заполняем таблицу my_summary_new необходимыми данными
mysql> RENAME TABLE my_summary TO my_summary_old, my_summary_new TO my_summary;
```

Если вы переименуете исходную таблицу `my_summary` в `my_summary_old` перед присвоением имени `my_summary` вновь созданной таблице, как сделали мы, то сможете хранить старую версию до тех пор, пока не придет время следующего перестроения. Целесообразно хранить ее, чтобы иметь возможность быстро выполнить откат в случае возникновения проблем с новой таблицей.

Материализованные представления

Многие системы управления базами данных, такие как Oracle или Microsoft SQL Server, предлагают возможность, называемую *материализованными представлениями*. Это представления, которые на самом деле вычисляются предварительно и хранятся в виде таблиц на диске и могут быть восстановлены и обновлены с помощью различных стратегий. MySQL по умолчанию не поддерживает эту возможность (мы подробно рассмотрим присущую ей поддержку представлений в главе 7). Однако вы можете реализовать материализованные представления самостоятельно, используя инструменты пакета Flexviews, созданные Джастином Сванхартом (Justin Swanhart) (http://code.google.com/p/flexviews/). Пакет Flexviews многофункциональнее самодельных решений и предлагает множество хороших функций, упрощающих создание и поддержку материализованных представлений. Он состоит из нескольких частей:

- средства для отслеживания измененных данных (Change Data Capture, CDC) — утилиты, считывающей двоичные журналы сервера и извлекающей релевантные изменения;

- набора хранимых процедур, которые помогают выявлять определения представлений и управлять ими;
- инструментов для изменения материализованных данных в базе данных.

В отличие от обычных методов хранения сводных и кэшированных таблиц пакет Flexviews может поэтапно пересчитывать содержимое материализованного представления, извлекая небольшие изменения в исходные таблицы. Это означает, что он может обновлять представление, не запрашивая исходные данные. Например, если вы создаете сводную таблицу, которая подсчитывает группы строк, а затем добавляете строку в исходную таблицу, Flexviews просто увеличивает соответствующую группу на единицу. Тот же прием работает для других агрегатных функций, таких как `SUM()` и `AVG()`. Здесь используется тот факт, что двоичный журнал на основе строк включает в себя образ строк до и после их обновления, поэтому Flexviews видит не только новое значение каждой строки, но и изменение по сравнению с предыдущей версией. При этом ему не требуется просматривать исходную таблицу. Вычисление на основе изменений намного эффективнее, чем чтение данных из исходной таблицы.

Из-за нехватки места мы не можем подробно рассказать, как пользоваться Flexviews. Тем не менее хотели бы дать небольшой обзор. Начните с ввода команды `SELECT`, описывающей данные, которые вы хотите получить из существующей базы данных. Эта команда может включать соединения и агрегации (`GROUP BY`). В Flexviews есть вспомогательный инструмент, преобразующий ваш SQL-запрос в вызовы API Flexviews. Затем Flexviews выполнит всю грязную работу по просмотру изменений в базе данных и преобразованию их в обновления для таблиц, которые хранят материализованное представление исходных таблиц. Теперь ваше приложение может просто запросить материализованное представление вместо таблиц, на основе которых оно было получено.

Инструментарий Flexviews обеспечивает хороший охват SQL и позволяет обрабатывать в том числе непростые выражения, причем даже сложно поверить, что такая обработка возможна за пределами сервера. Это делает его полезным для построения представлений на основе сложных SQL-выражений, поэтому вы можете заменить сложные запросы простым и быстрым запросом к материализованному представлению.

Таблицы счетчиков

Приложения, хранящие счетчики в таблицах, могут столкнуться с проблемами параллелизма при обновлении счетчиков. В веб-приложениях такие таблицы применяются очень часто. Вы можете использовать их для кэширования количества друзей пользователя, количества загрузок файла и т. п. Зачастую целесообразно создать отдельную таблицу для счетчиков — она невелика и работы будет немного. Применение отдельной таблицы поможет избежать удаления кэша запросов и позволит использовать кое-какие приемы, которые мы продемонстрируем в этом разделе.

Для простоты предположим, что у нас есть таблица счетчиков с одной строкой, в которой просто подсчитывается количество обращений к сайту:

```
mysql> CREATE TABLE hit_counter (
    -> cnt int unsigned not null
    -> ) ENGINE=InnoDB;
```

При каждом обращении к сайту счетчик обновляется:

```
mysql> UPDATE hit_counter SET cnt = cnt + 1;
```

Проблема состоит в том, что эта единственная строка фактически становится глобальным мьютексом для любой транзакции, которая обновляет счетчик. Транзакции оказываются сериализованными. Можно увеличить уровень параллелизма, создав несколько строк и обновляя случайно выбранную строку. Для этого необходимо выполнить следующие изменения:

```
mysql> CREATE TABLE hit_counter (
    ->    slot tinyint unsigned not null primary key,
    ->    cnt int unsigned not null
    -> ) ENGINE=InnoDB;
```

Предварительно заполните таблицу, добавив к ней 100 строк. Теперь запрос может просто выбрать случайную строку и обновить ее:

```
mysql> UPDATE hit_counter SET cnt = cnt + 1 WHERE slot = RAND() * 100;
```

Для извлечения статистики используйте агрегатный запрос:

```
mysql> SELECT SUM(cnt) FROM hit_counter;
```

Как правило, время от времени (например, один раз в день) требуется заново запускать счетчик. Для этого нужно слегка изменить схему:

```
mysql> CREATE TABLE daily_hit_counter (
    ->    day date not null,
    ->    slot tinyint unsigned not null,
    ->    cnt int unsigned not null,
    ->    primary key(day, slot)
    -> ) ENGINE=InnoDB;
```

В этом случае не требуется создавать строки заранее. Вместо этого можете использовать фразу `ON DUPLICATE KEY UPDATE`:

```
mysql> INSERT INTO daily_hit_counter(day, slot, cnt)
    ->    VALUES(CURRENT_DATE, RAND() * 100, 1)
    ->    ON DUPLICATE KEY UPDATE cnt = cnt + 1;
```

Если вы хотите для уменьшения размера таблицы уменьшить количество строк, напишите скрипт, который объединяет все результаты в строку `0` и удаляет остальные строки:

```
mysql> UPDATE daily_hit_counter as c
    ->    INNER JOIN (
    ->       SELECT day, SUM(cnt) AS cnt, MIN(slot) AS mslot
    ->       FROM daily_hit_counter
    ->       GROUP BY day
    ->    ) AS x USING(day)
    -> SET c.cnt  = IF(c.slot = x.mslot, x.cnt, 0),
    ->     c.slot = IF(c.slot = x.mslot, 0, c.slot);
mysql> DELETE FROM daily_hit_counter WHERE slot <> 0 AND cnt = 0;
```

Ускоренное чтение, замедленная запись

Чтобы ускорить работу запросов на чтение, часто приходится вводить дополнительные индексы, избыточные поля и даже кэшированные и сводные таблицы. Это добавляет работы по написанию запросов и обслуживающих заданий, зато повышает производительность: замедление записи компенсируется значительным ускорением операций чтения.

Однако это не единственная цена, которую вы платите за ускорение запросов на чтение. Увеличивается также сложность разработки как для операций чтения, так и для операций записи.

Ускорение работы команды ALTER TABLE

Производительность команды `ALTER TABLE` в MySQL может стать проблемой при работе с очень большими таблицами. Как правило, в этом случае MySQL создает пустую таблицу с заданной новой структурой, вставляет в нее все данные из старой таблицы и удаляет последнюю. Часто эта операция занимает много времени, особенно если памяти не очень много, а таблица велика и содержит немало индексов. Многие сталкивались с ситуацией, когда операция `ALTER TABLE` выполнялась несколько часов или даже дней.

В MySQL 5.1 и более поздних версиях добавлена поддержка некоторых типов онлайн-операций, которые не будут блокировать таблицу на весь период выполнения.

Последние версии InnoDB7[1] также поддерживают построение индексов путем сортировки, которая значительно ускоряет это действие и позволяет компактно разместить индексы.

Как правило, операция `ALTER TABLE` вызовет прерывание обслуживания в MySQL. Мы покажем некоторые приемы, позволяющие избежать этого, однако они предназначены для особых случаев. Нужно использовать либо операционные хитрости, например замену серверов и выполнение команды `ALTER` на серверах, не являющихся рабочими, либо метод теневой копии. Этот метод заключается в создании новой таблицы с заданной структурой. Затем вы переименовываете старую и новую таблицы и тем самым меняете их местами. В этом вам могут помочь специальные инструменты, например инструменты онлайнового изменения схемы, созданные командой, поддерживающей базы данных Facebook (https://launchpad.net/mysqlatfacebook), пакет openark, созданный Шломи Ноачем (Shlomi Noach) (http://code.openark.org/), и пакет Percona Toolkit (http://www.percona.com/software/). Если вы используете Flexviews (он рассматривался в разделе «Материализованные представления»), то можете выполнять неблокирующие изменения схемы и с помощью утилиты CDC.

[1] Это относится к так называемому плагину InnoDB, который является единственной версией InnoDB, существующей в MySQL 5.5 и более новых версиях. Подробнее об истории выпуска InnoDB говорится в главе 1.

Не все операции ALTER TABLE приводят к перестроению таблицы. Например, вы можете двумя способами (быстрым и медленным) изменить или удалить значение по умолчанию для столбца. Предположим, вы хотите изменить продолжительность проката фильма с трех до пяти дней. Далее представлен трудоемкий способ:

```
mysql> ALTER TABLE sakila.film
    -> MODIFY COLUMN rental_duration TINYINT(3) NOT NULL DEFAULT 5;
```

Команда SHOW STATUS показывает, что эта операция осуществляет 1000 считываний и 1000 вставок. Другими словами, она копирует старую таблицу в новую, несмотря на то что тип, размер и допустимость значения NULL не меняются.

Теоретически MySQL могла бы пропустить построение новой таблицы. В действительности значение по умолчанию для столбца хранится в .frm-файле таблицы, так что его можно изменить, не трогая ее саму. Однако пока эта оптимизация в MySQL не используется — любой оператор MODIFY COLUMN приводит к перестройке таблицы.

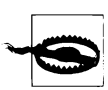
Но вы можете изменить значение по умолчанию для столбца с помощью команды ALTER COLUMN[1]:

```
mysql> ALTER TABLE sakila.film
    -> ALTER COLUMN rental_duration SET DEFAULT 5;
```

Эта команда модифицирует .frm-файл, но не затрагивает таблицу. Как следствие, она выполняется очень быстро.

Модификация одного лишь .frm-файла

Мы видели, что модификация .frm-файла таблицы происходит быстро и что MySQL иногда перестраивает таблицу, даже когда в этом нет необходимости. Если вы готовы рискнуть, можете убедить MySQL выполнить некоторые другие модификации без перестроения.



Прием, который мы собираемся продемонстрировать, официально не поддерживается, не документирован и может не работать. Используйте его на свой страх и риск. Советуем сначала сделать резервную копию данных!

Теоретически вы можете выполнять без перестроения таблицы следующие типы операций:

- удаление (но не добавление) атрибута столбца AUTO_INCREMENT;
- добавление, удаление или изменение констант ENUM и SET. Если вы удалите константу, а некоторые строки содержат это значение, запросы будут возвращать для него пустую строку.

[1] Команда ALTER TABLE позволяет модифицировать столбцы с помощью ALTER COLUMN, MODIFY COLUMN и CHANGE COLUMN. Все три варианта выполняют разные задания.

Суть приема заключается в создании `.frm`-файла для таблицы нужной структуры и копирования его на место существующего следующим образом.

1. Создайте пустую таблицу с точно такой же структурой, за исключением желаемой модификации (например, без добавленной константы `ENUM`)[1].
2. Выполните команду `FLUSH TABLES WITH READ LOCK`. Она закроет все используемые таблицы и предотвратит открытие любых таблиц.
3. Поменяйте местами `.frm`-файлы.
4. Выполните команду `UNLOCK TABLES`, чтобы снять блокировку чтения.

В качестве примера добавим константу к столбцу `rating` в таблице `sakila.film`. Сейчас столбец выглядит следующим образом:

```
mysql> SHOW COLUMNS FROM sakila.film LIKE 'rating';
+--------+-----------------------------------+------+-----+---------+-------+
| Field  | Type                              | Null | Key | Default | Extra |
+--------+-----------------------------------+------+-----+---------+-------+
| rating | enum('G','PG','PG-13','R','NC-17') | YES  |     | G       |       |
+--------+-----------------------------------+------+-----+---------+-------+
```

Мы добавим категорию `PG-14` для родителей, которые беспокоятся о том, что смотрят дети:

```
mysql> CREATE TABLE sakila.film_new LIKE sakila.film;
mysql> ALTER TABLE sakila.film_new
    -> MODIFY COLUMN rating ENUM('G','PG','PG-13','R','NC-17', 'PG-14')
    -> DEFAULT 'G';
mysql> FLUSH TABLES WITH READ LOCK;
```

Обратите внимание, что мы добавляем новое значение в конец списка констант. Если бы мы поместили его в середину, после `PG-13`, то изменились бы значения существующих данных: `R` превратились бы в `PG-14`, `NC-17` стали бы `R` и т. д.

Теперь из командной строки операционной системы поменяем местами `.frm`-файлы:

```
/var/lib/mysql/sakila# mv film.frm film_tmp.frm
/var/lib/mysql/sakila# mv film_new.frm film.frm
/var/lib/mysql/sakila# mv film_tmp.frm film_new.frm
```

Вернувшись в MySQL, мы можем разблокировать таблицу и увидеть, что изменения вступили в силу:

```
mysql> UNLOCK TABLES;
mysql> SHOW COLUMNS FROM sakila.film LIKE 'rating'\G
*************************** 1. row ***************************
Field: rating
 Type: enum('G','PG','PG-13','R','NC-17','PG-14')
```

Теперь удалим таблицу, которую создали для выполнения этой операции:

```
mysql> DROP TABLE sakila.film_new;
```

[1] Как видно из дальнейшего, после создания пустой таблицы ее модифицируют, исходя из желаемой структуры (например, добавляют константу ENUM). — *Примеч. пер.*

Быстрое построение индексов MyISAM

Обычно для эффективной загрузки таблиц MyISAM применяют следующий прием: сначала блокируют ключи, затем загружают данные и повторно активизируют ключи:

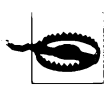
```
mysql> ALTER TABLE test.load_data DISABLE KEYS;
-- загрузка данных
mysql> ALTER TABLE test.load_data ENABLE KEYS;
```

Это работает, поскольку MyISAM позволяет отложить построение ключей до тех пор, пока все данные не загрузятся, после чего она может построить индексы путем сортировки. Такая операция происходит намного быстрее и дает дефрагментированное компактное индексное дерево[1].

К сожалению, данный прием не работает для уникальных индексов, поскольку модификатор `DISABLE KEYS` можно применить только к неуникальным. MyISAM строит уникальные индексы в памяти и проверяет их уникальность при загрузке каждой строки. Как только размер индекса превышает объем доступной памяти, загрузка становится чрезвычайно медленной.

В современных версиях InnoDB можно использовать аналогичные приемы, основанные на возможности быстрого создания онлайновых индексов InnoDB. Для этого требуется удалить все неуникальные индексы, добавить новый столбец и возвратить удаленные индексы. Percona Server автоматически поддерживает этот способ.

Как и в случае с трюком, проделанным с `ALTER TABLE` в предыдущем разделе, вы можете ускорить этот процесс, если готовы немного больше поработать и чуть-чуть рискнуть. Этот прием можно применять для загрузки данных из резервных копий, например, когда вы знаете, что все значения корректны и нет необходимости проверять уникальность.



Данный прием также не поддерживается официально и не документирован. Используйте его на свой страх и риск, не забывая сначала сделать резервную копию данных.

Выполните следующие действия.

1. Создайте таблицу с желаемой структурой, но без всяких индексов.
2. Загрузите данные в таблицу, чтобы построить `.MYD`-файл.
3. Создайте пустую таблицу с желаемой структурой, но в этот раз с индексами. Будут сгенерированы необходимые `.frm`- и `.MYD`-файлы.
4. Сбросьте таблицы с блокировкой чтения.
5. Переименуйте `.frm`- и `.MYI`-файлы второй таблицы, чтобы MySQL использовала их для первой таблицы.
6. Снимите блокировку чтения.

[1] MyISAM также будет строить индексы путем сортировки, если вы примените команду LOAD DATA INFILE, а таблица будет пуста.

7. Используйте команду `REPAIR TABLE` для создания индексов таблицы. Эта команда построит все индексы, в том числе уникальные, путем сортировки.

Для очень больших таблиц эта процедура может оказаться гораздо более быстрой.

Итоги главы

Хорошо спроектированная схема довольно универсальна, однако у MySQL есть особенности реализации, которые стоит учитывать. В двух словах: стоит хранить данные в настолько маленьких и простых таблицах, насколько это возможно. MySQL любит простоту. Кроме того, учитывайте, что с вашей базой данных придется работать другим людям.

- При проектировании старайтесь избегать экстремальных ситуаций, таких как использование схемы, из-за особенностей которой придется писать чрезвычайно сложные запросы или таблицы с уймой столбцов. (Уйма — это где-то между множеством и несметным количеством.)
- Используйте небольшие, простые и подходящие по контексту типы данных, применяйте `NULL`, только если это действительно правильный способ моделирования ваших данных.
- Старайтесь использовать одни и те же типы данных для хранения похожих или связанных значений, особенно если они будут работать в качестве условий при соединениях.
- Следите за строками переменной длины, поскольку они могут привести к неудачному выделению памяти на целую строку для временных таблиц и сортировок.
- По возможности задавайте для идентификаторов целочисленные значения.
- Избегайте устаревших методик MySQL, таких как определение точности для чисел с плавающей запятой или ширины отображения для целых чисел.
- Будьте осторожны с типами `ENUM` и `SET`. Они удобны, но ими не стоит злоупотреблять, тем более что иногда они мудрены. Типа `BIT` лучше избегать.

Нормализация — это хорошо, но иногда денормализация (дублирование данных в большинстве случаев) бывает действительно необходимой и полезной. В следующей главе мы приведем больше примеров, свидетельствующих об этом. Полезными могут оказаться также предварительные вычисления, кэширование и создание сводных таблиц. Пакет Flexviews, разработанный Джастином Сванхартом, помогает работать со сводными таблицами.

Команда `ALTER TABLE` может доставить неприятности, поскольку в большинстве случаев она блокирует и перестраивает всю таблицу. Мы показали ряд обходных решений для конкретных случаев. В общем же случае вам придется использовать другие приемы, такие как выполнение команды `ALTER` на копии, а затем ее расширение для главной таблицы. Об этом мы подробно расскажем чуть позже.

5 Повышение производительности с помощью индексирования

Индексы, в MySQL также называемые *ключами*, представляют собой структуры данных, которые подсистемы хранения используют для быстрого нахождения строк. Кроме того, у них есть еще несколько достоинств, которые мы опишем в этой главе.

Индексы критически значимы для достижения хорошей производительности и становятся все более важными по мере роста объема данных. Небольшие слабо загруженные базы данных зачастую могут хорошо работать даже без правильно построенных индексов, но по мере роста объема хранимой в базе информации производительность может очень быстро упасть[1]. К сожалению, на практике об индексах часто забывают или плохо понимают их смысл, поэтому плохое индексирование является главной причиной проблем с производительностью. Так что мы решили рассмотреть этот материал в первой части книги — даже раньше, чем обсуждать оптимизацию запросов.

Оптимизация индекса — это, пожалуй, самый мощный способ повысить производительность запросов. Индексы могут увеличить производительность на много порядков, а оптимальные индексы иногда способны повысить ее примерно на два порядка больше, чем просто хорошие индексы. Для создания действительно оптимальных индексов часто требуется переписать запросы, поэтому текущая и следующая главы тесно связаны между собой.

---

[1] В этой главе, если не указано иное, предполагается, что используются обычные жесткие диски. Твердотельные накопители имеют другие характеристики производительности, и в этой книге мы рассмотрим это различие. Принципы индексирования не меняются, но потери, которых мы стремимся избежать, у твердотельных дисков не такие большие, как у обычных.

Основы индексирования

Самый простой способ понять, как работает индекс в MySQL, — представить себе алфавитный указатель в книге. Чтобы определить, где в книге обсуждается конкретная тема, вы в алфавитном указателе находите термин и номер страницы, на которой он упоминается.

Система MySQL использует индексы схожим образом. Она ищет значение в структурах данных индекса и, обнаружив соответствие, может перейти к самой строке. Предположим, выполняется следующий запрос:

```
mysql> SELECT first_name FROM sakila.actor WHERE actor_id = 5;
```

По столбцу `actor_id` построен индекс, поэтому MySQL будет использовать его для поиска строк, в которых значение поля `actor_id` равняется `5`. Другими словами, система ищет в индексе значение и возвращает все содержащие его строки.

В индекс включены данные из одного или нескольких столбцов таблицы. Если индекс построен по нескольким столбцам, то их порядок очень важен, поскольку MySQL может эффективно выполнять поиск только по крайней левой части индекса. Как вы увидите в дальнейшем, создание индекса по двум столбцам — это совсем не то же самое, что создание двух отдельных индексов, каждый по одному столбцу.

Должен ли я это читать, если у меня ORM?

Краткий ответ: да, вам все равно нужно узнать об индексации, даже если вы полагаетесь на объектно-реляционное отображение (object-relational mapping, ORM).

Инструменты ORM выполняют логически и синтаксически корректные запросы (как правило), но они редко создают запросы, благоприятные для индексирования, за исключением случаев, когда вы используете их только для основных типов запросов, таких как поиск по первичным ключам. Не стоит ожидать, что ваш инструмент ORM, каким бы сложным он ни был, справится с тонкостями и трудностями индексирования. Если вы не согласны, прочитайте эту главу до конца! Иногда даже опытному человеку непросто разобраться со всеми вариантами, что уж говорить об ORM.

Типы индексов

Существует множество типов индексов, каждый из которых лучше подходит для достижения той или иной цели. Индексы реализуются на уровне подсистем хранения, а не на уровне сервера. Таким образом, они не стандартизованы: индексирование работает по-разному в зависимости от подсистемы и далеко не все подсистемы поддерживают все типы индексов. Даже если несколько подсистем хранения поддерживают определенный тип индекса, его внутренняя реализация может существенно различаться.

Помня об этом, рассмотрим типы индексов, с которыми в настоящее время умеет взаимодействовать MySQL, а также поговорим об их достоинствах и недостатках.

Индексы, упорядоченные на основе B-дерева

Когда говорят об индексе без упоминания типа, обычно имеют в виду *индексы, упорядоченные на основе B-дерева*, в которых для хранения данных используется структура, называемая B-деревом[1]. Большинство подсистем хранения в MySQL поддерживают этот тип индекса. Исключение составляет подсистема Archive, в которой до версии MySQL 5.1 индексирование отсутствовало. Начиная с этой версии, появилась возможность строить индекс по одному столбцу типа `AUTO_INCREMENT`.

Мы используем для этих индексов термин «B-дерево» потому, что именно так MySQL называет их в `CREATE TABLE` и других командах. Однако на внутреннем уровне в подсистемах хранения могут использоваться совершенно иные структуры данных. Например, в подсистеме NDB Cluster для таких индексов применяется структура данных «T-дерево», хотя они по-прежнему называются `BTREE`, а InnoDB использует тип «B + дерево». Однако в данной книге мы не будем рассматривать различия в структуре и алгоритмах.

Подсистемы хранения по-разному используют индексы, упорядоченные на основе B-дерева, и это может влиять на производительность. Например, в MyISAM применяется техника сжатия префикса, позволяющая уменьшить размер индекса, а InnoDB индексы не сжимает. Кроме того, индексы MyISAM ссылаются на индексированные строки по их физическому адресу на диске, а индексы InnoDB — по значениям первичного ключа. Каждый вариант имеет свои достоинства и недостатки.

Общая идея B-дерева заключается в том, что значения хранятся по порядку и все листья-страницы находятся на одинаковом расстоянии от корня. На рис. 5.1 показано абстрактное изображение индекса, упорядоченного на основе B-дерева, приблизительно соответствующее работе индексов InnoDB. MyISAM использует другую структуру, но принципы похожи.

Индекс, упорядоченный на основе B-дерева, ускоряет доступ к данным, поскольку подсистеме хранения не нужно сканировать всю таблицу для поиска искомых данных. Вместо этого она начинает с корневого узла (не показанного на рис. 5.1). В корневом узле имеется массив указателей на дочерние узлы, и подсистема хранения переходит по этим указателям. Для нахождения подходящего указателя она просматривает значения в узловых страницах, которые определяют верхнюю и нижнюю границы значений в дочерних узлах. В итоге подсистема хранения либо определяет, что искомого значения не существует, либо благополучно достигает нужного листа.

Листья-страницы представляют собой особый случай, поскольку в них находятся указатели на индексированные данные, а не на другие страницы. (В различных подсистемах хранения применяются разные типы указателей на данные.) На рис. 5.1

[1] Во многих подсистемах хранения на самом деле используются индексы типа «B + дерево», в которых каждый лист содержит указатель на следующий лист для ускорения обхода дерева по диапазону значений. Более подробное описание индексов со структурой B-дерева можно найти в литературе по информатике.

показаны только одна узловая страница и соответствующие ей листья, но между корнем и листьями может быть много уровней узловых страниц. Глубина дерева зависит от размера таблицы.

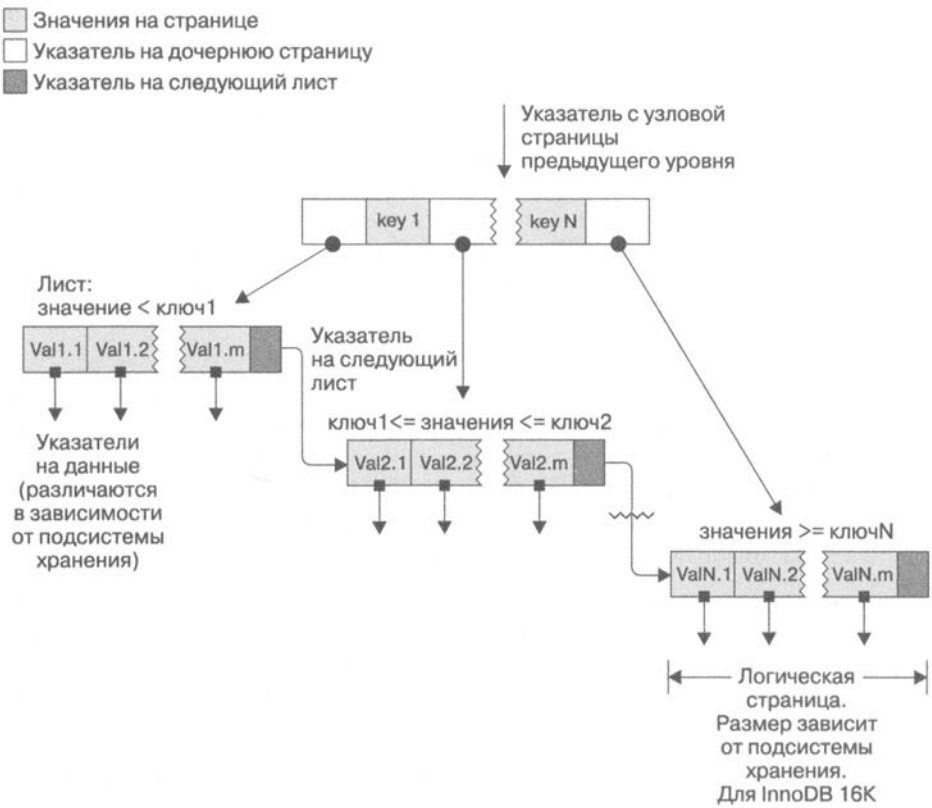


Рис. 5.1. Индекс, упорядоченный на основе структуры B-дерева (технически «B + дерево»)

Поскольку в индексах, упорядоченных на основе структуры B-дерева, индексированные столбцы хранятся в упорядоченном виде, они полезны для поиска по диапазону данных. При спуске вниз по дереву индекса, построенного по текстовому полю, значения будут перебираться в алфавитном порядке, поэтому поиск, например, всех людей, чьи фамилии начинаются с букв от «И» до «К», окажется эффективным.

Предположим, у нас есть следующая таблица:

```
CREATE TABLE People (
    last_name  varchar(50)    not null,
    first_name varchar(50)    not null,
    dob        date           not null,
    gender     enum('m', 'f') not null,
    key(last_name, first_name, dob)
);
```

Индекс будет содержать значения из столбцов `last_name`, `first_name` и `dob` для каждой строки в таблице. На рис. 5.2 показано, как организовано хранение данных в индексе.

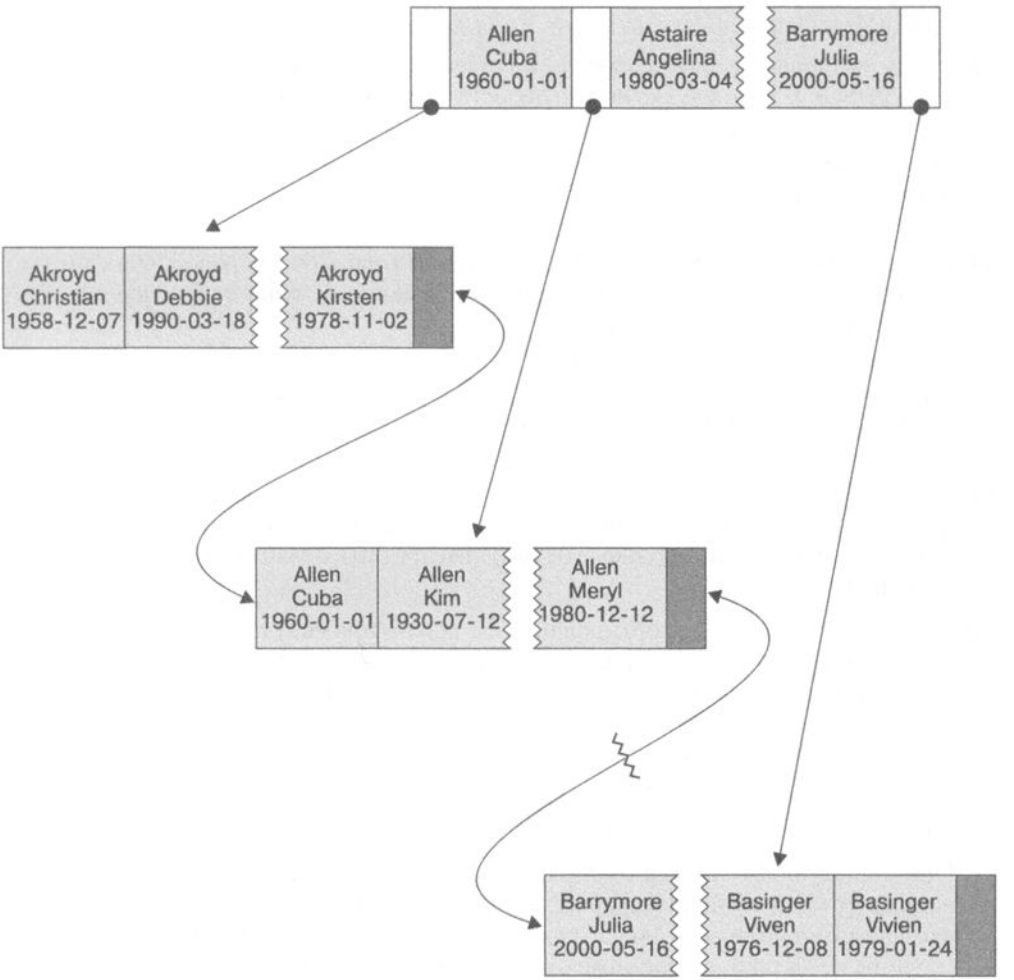


Рис. 5.2. Пример расположения записей в индексе, упорядоченном на основе B-дерева (технически «B + дерево»)

Обратите внимание на то, что в индексе значения упорядочены в том же порядке, в котором столбцы указаны в команде `CREATE TABLE`. Посмотрите на две последние записи: два человека с одинаковыми фамилией и именем, но разными датами рождения, отсортированы именно по этому параметру.

Типы запросов, в которых может использоваться индекс, упорядоченный на основе B-дерева. Индексы B-дерева хорошо работают при поиске по полному значению ключа, диапазону ключей или префиксу ключа. Они полезны только тогда, когда

используется крайний левый префикс ключа[1]. Индекс, показанный в предыдущем разделе, будет полезен для следующих типов запросов.

- *Поиск по полному значению.* При поиске по полному значению ключа задаются критерии для всех столбцов, по которым построен индекс. Например, индекс позволит найти человека по имени Куба Аллен, родившегося 1 января 1960 года.
- *Поиск по крайнему левому префиксу.* Индекс позволит найти всех людей с фамилией Аллен. В этом случае используется только первый столбец индекса.
- *Поиск по префиксу столбца.* Можно искать соответствие по началу значения столбца. Рассматриваемый индекс позволит найти всех людей, чьи фамилии начинаются с буквы «А». В этом случае используется только первый столбец индекса.
- *Поиск по диапазону значений.* Индекс позволит найти всех людей, чьи фамилии находятся между Алленом и Бэрримором. В данном случае также используется только первый столбец индекса.
- *Поиск по полному совпадению одной части и диапазону в другой части.* Индекс позволит найти всех людей по фамилии Аллен, чьи имена начинаются с буквы «К» (Ким, Карл и т. д.). В данном случае выполняется поиск по полному значению в столбце `last_name` и поиск по диапазону значений в столбце `first_name`.
- *Запросы только по индексу.* Индексы, упорядоченные на основе B-дерева, обычно могут поддерживать запросы только по индексу, то есть обращение именно к индексу, а не к самой строке. Мы обсудим эту оптимизацию в разделе «Покрывающие индексы».

Поскольку узлы дерева отсортированы, их можно применять как для поиска значений, так и для запросов с фразой `ORDER BY` (поиска значений в отсортированных строках). В общем случае, если B-дерево позволяет найти строку по определенному критерию, его можно использовать и для сортировки строк по тому же критерию. Именно поэтому индекс будет полезен для запросов с фразой `ORDER BY`, в которой сортировка соответствует перечисленным видам поиска.

Для применения индексов, упорядоченных на основе B-дерева, есть ряд ограничений.

- Они бесполезны, если в критерии поиска не указан крайний левый из индексированных столбцов. Например, рассматриваемый индекс не поможет найти людей с именем Билл или всех людей, родившихся в определенный день, поскольку эти столбцы не являются крайними слева в индексе. Такой индекс непригоден и для поиска людей, чьи фамилии заканчиваются на конкретную букву.
- Нельзя пропускать столбцы индекса. То есть невозможно найти всех Смитов, родившихся в конкретный день. Если не указать значение столбца `first_name`, MySQL сможет использовать только первый столбец индекса.
- Подсистема хранения не может оптимизировать поиск по столбцам, находящимся правее первого столбца, по которому осуществляется поиск в заданном

[1] Это особенность MySQL, точнее, ее версий. В других базах данных можно искать не только по начальной части ключа, хотя обычно использование префикса эффективнее. Вероятно, в будущем такая возможность появится и в MySQL. Как обойти это ограничение, покажем позже в этой главе.

диапазоне. Например, для запроса с условием `WHERE last_name="Smith" AND first_name LIKE J%' AND dob='1976-12-23'` будут задействованы только первые два столбца индекса, поскольку `LIKE` задает диапазон условия (однако сервер может использовать оставшиеся столбцы для других целей). Для столбца, имеющего ограниченный набор значений, эту проблему можно обойти, указав условие равенства вместо условия, задающего диапазон. Далее в текущей главе, в кейсе по индексированию, мы приведем подробные примеры на эту тему.

Теперь вы поняли, почему так важен порядок столбцов: все эти ограничения так или иначе связаны именно с ним. Для достижения оптимальной производительности может потребоваться создать индексы с одними и теми же столбцами, стоящими в различном порядке.

Некоторые из названных ограничений не связаны с индексами, упорядоченными на основе B-дерева, а являются следствием того, как оптимизатор запросов MySQL и подсистемы хранения используют индексы. Возможно, некоторые ограничения в дальнейшем будут устранены.

Хеш-индексы

Хеш-индекс строится на основе хеш-таблицы и полезен только для точного поиска, использующего все столбцы индекса[1]. Для каждой строки подсистема хранения вычисляет *хеш-код* индексированных столбцов — сравнительно короткое значение, которое, скорее всего, будет различным для строк с разными значениями ключей. В индексе хранятся хеш-коды и указатели на соответствующие строки в хеш-таблице.

В MySQL только подсистема хранения Memory поддерживает явные хеш-индексы. Этот тип индекса по умолчанию применяется для таблиц типа Memory, хотя они могут использовать и индексы, упорядоченные на основе B-дерева. Подсистема Memory поддерживает неуникальные хеш-индексы, что необычно для сферы баз данных. Если у нескольких значений один и тот же хеш-код, то в индексе связанный список указателей на их строки будет храниться в одной и той же записи хеш-таблицы.

Приведем пример. Предположим, имеется таблица:

```
CREATE TABLE testhash (
    fname VARCHAR(50) NOT NULL,
    lname VARCHAR(50) NOT NULL,
    KEY USING HASH(fname)
) ENGINE=MEMORY;
```

содержащая следующие данные:

```
mysql> SELECT * FROM testhash;
+--------+-----------+
| fname  | lname     |
+--------+-----------+
| Arjen  | Lentz     |
| Baron  | Schwartz  |
| Peter  | Zaitsev   |
| Vadim  | Tkachenko |
+--------+-----------+
```

[1] Более подробную информацию о хеш-таблицах можно найти в литературе по информатике.

Теперь допустим, что для индексирования используется воображаемая хеш-функция `f()`, которая возвращает следующие значения (это просто примеры, а не реальные значения):

```
f('Arjen')= 2323
f('Baron')= 7437
f('Peter')= 8784
f('Vadim')= 2458
```

Структура данных индекса будет выглядеть следующим образом:

Ячейка	Значение
2323	Указатель на строку 1
2458	Указатель на строку 4
7437	Указатель на строку 2
8784	Указатель на строку 3

Обратите внимание, что ячейки упорядочены, а строки — нет. Теперь, когда мы выполним следующий запрос:

```
mysql> SELECT lname FROM testhash WHERE fname='Peter';
```

MySQL вычислит хеш-код значения `'Peter'` и использует его для поиска указателя в индексе. Поскольку `f('Peter')` = `8784`, MySQL будет искать в индексе значение `8784` и найдет указатель на строку 3. Заключительным шагом будет сравнение значения в строке 3 со значением `'Peter'` с целью убедиться в том, что это действительно искомая строка.

Поскольку в индексе хранятся только короткие хеш-коды, то хеш-индексы очень компактны. Поэтому поиск обычно выполняется молниеносно. Однако у хеш-индексов есть ряд ограничений.

- Поскольку индекс содержит только хеш-коды и указатели на строки, а не сами значения, MySQL не может использовать данные в индексе, чтобы избежать чтения строк. К счастью, доступ к строкам, находящимся в памяти, можно получить очень быстро, так что обычно это не снижает производительности.
- MySQL не может использовать хеш-индексы для сортировки, поскольку строки в этой системе не отсортированы.
- Хеш-индексы не поддерживают поиск по частичному ключу, поскольку хеш-коды вычисляются для всего индексируемого значения. Иначе говоря, если имеется индекс по столбцам `(A,B)`, а в разделе `WHERE` упоминается только столбец `A`, индекс не поможет.
- Хеш-индексы поддерживают лишь сравнения на равенство, то есть использование операторов `=`, `IN()` и `<=>` (обратите внимание, что `<>` и `<=>` — разные операторы). Они не могут ускорить поиск по диапазону, например, `WHERE price > 100`.
- Доступ к данным в хеш-индексе можно получить очень быстро, если нет большого количества коллизий (нескольких значений с одним и тем же хеш-кодом). При их наличии подсистема хранения вынуждена проходить по каждому указателю

на строку, хранящемуся в связанном списке, и сравнивать значение в этой строке с искомым.

- Некоторые операции обслуживания индекса могут оказаться медленными, если количество коллизий велико. Например, если вы создаете хеш-индекс по столбцу с очень маленькой селективностью (много коллизий), а затем удаляете строку из таблицы, то на поиск указателя на эту строку в индексе может быть затрачено много времени. Подсистеме хранения придется проверять каждую строку в связанном списке для этого ключа, чтобы найти и убрать ссылку на ту единственную строку, которую вы удалили.

Из-за этих ограничений хеш-индексы оказываются полезными только в особых случаях. Однако если они соответствуют потребностям приложения, то могут значительно повысить производительность. В качестве примера можно привести хранилища данных, где классическая схема «звезда» подразумевает много соединений со справочными таблицами. Хеш-индексы — именно то, что требуется для подобного случая.

Кроме явных хеш-индексов в подсистеме хранения Memory, подсистема хранения NDB Cluster поддерживает уникальные хеш-индексы. Их функциональность специфична только для этой подсистемы, которую мы не описываем в книге.

Подсистема хранения InnoDB поддерживает так называемые *адаптивные хеш-индексы*. Когда InnoDB замечает, что доступ к некоторым значениям индекса требуется очень часто, она строит для них хеш-индекс в памяти вдобавок к индексам на основе B-дерева. Это придает индексам, упорядоченным на основе B-дерева, некоторые свойства хеш-индексов, например появляется возможность очень быстрого поиска. Это автоматический процесс, вы не можете ни управлять им, ни настраивать его. Хотя можете отключить.

Построение собственных хеш-индексов. Если подсистема хранения не поддерживает хеш-индексы, то вы можете эмулировать их самостоятельно, подобно тому как это делает InnoDB. Тем самым сможете использовать некоторые достоинства хеш-индексов, например небольшие размеры индекса при очень длинных ключах.

Все просто: вдобавок к стандартным индексам, упорядоченным на основе B-дерева, создайте псевдохеш-индекс. Он не совсем идентичен настоящему хеш-индексу, поскольку для поиска по-прежнему станет использоваться индекс на основе B-дерева. Однако будет выполняться поиск хеш-кодов ключей вместо самих ключей. Все, что вам нужно сделать, — вручную указать хеш-функцию в запросе в разделе `WHERE`.

Примером подобного подхода, причем хорошо работающего, является поиск адресов URL. Индексы, упорядоченные на основе B-дерева по адресам URL, обычно оказываются огромными, поскольку сами URL длинные. Рядовой запрос к таблице адресов URL выглядит примерно так:

```
mysql> SELECT id FROM url WHERE url="http://www.mysql.com";
```

Но если удалить индекс по столбцу `url` и добавить в таблицу индексированный столбец `url_crc`, то запрос можно переписать следующим образом:

```
mysql> SELECT id FROM url WHERE url="http://www.mysql.com"
    -> AND url_crc=CRC32("http://www.mysql.com");
```

Этот подход хорошо работает, поскольку оптимизатор запросов MySQL замечает, что существует небольшой высокоизбирательный индекс по столбцу `url_crc`, и ищет в индексе элементы с этим значением (в данном случае **1560514994**). Даже если несколько строк имеют одно и то же значение `url_crc`, очень легко найти их с помощью быстрого целочисленного сравнения, а затем отыскать среди них ту, которая точно соответствует полному адресу URL. Альтернативой этому подходу является индексирование URL как строки, что существенно медленнее.

Один из недостатков этого подхода состоит в необходимости поддерживать хеш-значения. Вы можете делать это вручную или с помощью триггеров — в MySQL 5.0 и более поздних версиях. В следующем примере показано, как триггеры могут помочь поддерживать столбец `url_crc` при вставке и обновлении значений. Прежде всего создаем таблицу:

```
CREATE TABLE pseudohash (
    id int unsigned NOT NULL auto_increment,
    url varchar(255) NOT NULL,
    url_crc int unsigned NOT NULL DEFAULT 0,
    PRIMARY KEY(id)
 );
```

Теперь создаем триггеры. Мы временно изменяем разделитель команд, чтобы можно было использовать точку с запятой в качестве разделителя внутри триггера:

```
DELIMITER //

CREATE TRIGGER pseudohash_crc_ins BEFORE INSERT ON pseudohash FOR EACH ROW BEGIN
SET NEW.url_crc=crc32(NEW.url);
END;
//

CREATE TRIGGER pseudohash_crc_upd BEFORE UPDATE ON pseudohash FOR EACH ROW BEGIN
SET NEW.url_crc=crc32(NEW.url);
END;
//

DELIMITER ;
```

Осталось проверить, что триггер действительно изменяет хеш-код:

```
mysql> INSERT INTO pseudohash (url) VALUES ('http://www.mysql.com');
mysql> SELECT * FROM pseudohash;
+----+----------------------+------------+
| id | url                  | url_crc    |
+----+----------------------+------------+
|  1 | http://www.mysql.com | 1560514994 |
+----+----------------------+------------+
mysql> UPDATE pseudohash SET url='http://www.mysql.com/' WHERE id=1;
mysql> SELECT * FROM pseudohash;
+----+----------------------+------------+
| id | url                  | url_crc    |
+----+----------------------+------------+
|  1 | http://www.mysql.com/ | 1558250469 |
+----+----------------------+------------+
```

При таком подходе не следует использовать хеш-функции `SHA1()` или `MD5()`. Они возвращают очень длинные строки, которые занимают много места и замедляют

сравнение. Это мощные криптографические функции, спроектированные для почти гарантированного устранения противоречивых запросов, а не для достижения текущих целей. Простые хеш-функции могут обеспечить приемлемый уровень противоречивых запросов с более высокой производительностью.

Если в таблице много строк и функция `CRC32()` дает слишком много коллизий, реализуйте собственную 64-разрядную хеш-функцию. Она должна возвращать целое число, а не строку. Один из способов реализации 64-разрядной хеш-функции состоит в использовании только части значения, возвращаемого функцией `MD5()`. По-видимому, это менее эффективно, чем написание собственной, то есть пользовательской функции (см. главу 7), зато требует минимума усилий:

```
mysql> SELECT CONV(RIGHT(MD5('http://www.mysql.com/'), 16), 16, 10) AS HASH64;
+--------------------+
| HASH64             |
+--------------------+
| 9761173720318281581 |
+--------------------+
```

Обработка коллизий хеширования. При поиске значения по его хеш-коду следует включать в раздел `WHERE` и само искомое значение:

```
mysql> SELECT id FROM url WHERE url_crc=CRC32("http://www.mysql.com")
    ->    AND url="http://www.mysql.com";
```

Следующий запрос будет работать неправильно, поскольку, если для другого URL функция `CRC32()` возвращает значение `1560514994`, то запрос вернет обе строки:

```
mysql> SELECT id FROM url WHERE url_crc=CRC32("http://www.mysql.com");
```

Вероятность хеш-коллизий растет значительно быстрее, чем можно предположить, из-за так называемого парадокса дней рождения. Функция `CRC32()` возвращает 32-разрядное целое число, поэтому вероятность коллизии достигает 1 % уже при 93 000 значений. Чтобы проиллюстрировать это, мы загрузили в таблицу все слова из файла `/usr/share/dict/words` вместе с их значениями `CRC32()`. В результате получилось 98 569 строк. В этом наборе данных уже есть одна коллизия! Из-за нее следующий запрос возвращает несколько строк:

```
mysql> SELECT word, crc FROM words WHERE crc = CRC32('gnu');
+---------+------------+
| word    | crc        |
+---------+------------+
| codding | 1774765869 |
| gnu     | 1774765869 |
+---------+------------+
```

Правильный запрос будет выглядеть так:

```
mysql> SELECT word, crc FROM words WHERE crc = CRC32('gnu')AND word = 'gnu';
+------+------------+
| word | crc        |
+------+------------+
| gnu  | 1774765869 |
+------+------------+
```

Чтобы избежать проблем с противоречивыми запросами, следует указать в разделе `WHERE` оба условия. Если такие запросы не являются проблемой — например, потому,

что вы делаете статистические запросы и вам не нужны точные результаты, — то запрос можно упростить и вместе с тем повысить его эффективность, оставив в разделе `WHERE` только сравнение со значением функции `CRC32()`. Вы также можете использовать функцию `FNV64()` из пакета Percona Server, которую можно установить как плагин в любую версию MySQL. Эта функция очень быстрая, возвращает 64-разрядное число и гораздо меньше подвержена ошибкам, чем `CRC32()`.

Пространственные индексы (R-дерево)

MyISAM поддерживает пространственные индексы, которые можно использовать с частичными типами, такими как `GEOMETRY`. В отличие от индексов, упорядоченных на основе B-дерева, при работе с пространственными индексами не требуется, чтобы в разделе `WHERE` указывался крайний левый из индексированных столбцов. Они одновременно индексируют данные по всем столбцам. В результате при поиске может эффективно использоваться любая комбинация столбцов. Однако для этого необходимо задействовать функции GIS, такие как `MBRCONTAINS()`. MySQL поддерживает их не лучшим образом, поэтому их редко применяют. Рабочее решение для GIS в реляционной СУБД с открытым исходным кодом — это PostGIS в PostgreSQL.

Полнотекстовые индексы

Полнотекстовый (`FULLTEXT`) индекс представляет собой специальный тип индекса, который ищет ключевые слова в тексте вместо того, чтобы прямо сравнивать искомое значение со значениями в индексе. Полнотекстовый поиск кардинально отличается от других типов поиска. Он позволяет использовать стоп-слова, морфологический поиск, учет множественного числа, а также булев поиск. Он гораздо больше напоминает работу поисковых систем, чем простое сравнение с критерием в разделе `WHERE`.

Наличие полнотекстового индекса по столбцу не делает индекс, упорядоченный на основе B-дерева по этому столбцу, менее полезным. Полнотекстовые индексы предназначены для операций `MATCH AGAINST`, а не для обычных операций с фразой `WHERE`.

Более подробно полнотекстовое индексирование мы обсудим в главе 7.

Прочие типы индексов

В некоторых подсистемах хранения сторонних производителей для индексов применяются разные типы структур данных. Например, TokuDB использует индексы, упорядоченные на основе фрактального дерева. Это новая структура данных, у которой есть ряд преимуществ перед индексами, упорядоченными на основе B-дерева, и нет многих их недостатков. В этой главе мы будем рассматривать темы, связанные с InnoDB, включая кластерные и покрывающие индексы. В большинстве случаев то, что мы скажем об InnoDB, может относиться и к TokuDB. ScaleDB использует базисное дерево, а другие системы, такие как InfiniDB или Infobright, имеют собственные специальные структуры данных для оптимизации запросов.

Преимущества индексов

Индексы позволяют серверу быстро перемещаться в искомую позицию в таблице, но этим их достоинства не исчерпываются. Как вы, вероятно, уже поняли, индексы имеют несколько дополнительных преимуществ, которые основаны на свойствах структур данных, используемых для их создания. Индексы, упорядоченные на основе B-дерева, то есть самый распространенный их тип, работают, храня данные в отсортированном порядке, а MySQL может использовать их для запросов с фразами **ORDER BY** и **GROUP BY**. Поскольку данные предварительно отсортированы, индекс, упорядоченный на основе B-дерева, хранит связанные значения близко друг к другу. Наконец, индекс фактически хранит копию значений, как следствие, некоторые запросы могут быть выполнены только за счет индекса. Из этих свойств вытекают три основных преимущества.

1. Индексы уменьшают объем данных, которые сервер должен просмотреть для нахождения искомого значения.
2. Индексы помогают серверу избежать сортировки и временных таблиц.
3. Индексы превращают произвольный ввод/вывод в последовательный.

Рассмотрению индексов на самом деле можно посвятить целую книгу. Тем, кто хотел бы изучить этот вопрос подробнее, можем порекомендовать книгу Тапио Лахденмаки (Tapio Lahdenmaki) и Майка Лича (Mike Leach) *Relational Database Index Design and the Optimizers* (издательство Wiley). Из нее вы, помимо прочего, узнаете, как определить выгоду от создания и применения индексов и оценить скорость запросов.

Кроме того, в книге Лахденмаки и Лича предложена трехзвездочная система оценки того, насколько индекс подходит для запроса. Индекс получает одну звезду, если размещает связанные строки рядом друг с другом, две — если его строки сортируются в требуемом запросом порядке, и три — если содержит все столбцы, необходимые для запроса.

Мы вернемся к этим принципам далее в текущей главе.

Является ли индекс наилучшим решением?

Индекс не всегда является подходящим инструментом. Не будем здесь вдаваться в подробности, скажем только, что индексы наиболее эффективны, когда помогают подсистеме хранения находить строки, сокращая объем выполненной работы. Если таблица очень маленькая, часто проще бывает прочитать в ней все строки. Для средних и больших таблиц применение индексов может быть очень эффективным. Когда таблица огромная, затраты на индексирование, а также работы, необходимые для использования индексов, могут начать суммироваться. В таких случаях, скорее всего, нужно выбрать методику, когда определяются не отдельные строки, а группы строк, на которые ориентирован запрос. Для этой цели можно использовать секционирование (см. главу 7).

Если у вас много таблиц, стоит создать таблицу метаданных для хранения некоторых характеристик, представляющих интерес для запросов. Предположим, что у вас

есть многопользовательское приложение, которое обращается к данным, хранящимся в разных таблицах. Вы выполняете запросы, которые агрегируют данные по строкам. В этом случае можете записать, данные о каких пользователях хранятся в той или иной таблице, и просто игнорировать таблицы, в которых нет информации о конкретном пользователе. Этот прием будет полезен только при чрезвычайно больших объемах работ. Фактически что-то подобное делает Infobright. При терабайтовых объемах поиск отдельных строк не имеет смысла — индексы заменяются метаданными о конкретной группе блоков.

Стратегии индексирования для достижения высокой производительности

Создание правильных индексов и их корректное использование важны для достижения высокой производительности запросов. Мы рассказали о различных типах индексов, об их достоинствах и недостатках. Теперь посмотрим, как на практике максимально продуктивно использовать индексы.

Существует много способов эффективного выбора и использования индексов, поскольку предусмотрено множество специализированных оптимизаций и вариантов специализированного поведения. Умение определять, что и когда применять, а также оценивать влияние вашего выбора на производительность системы — это навык, приходящий с опытом. Следующие разделы помогут вам понять, как эффективно использовать индексы.

Изоляция столбца

Мы неоднократно видели запросы, которые разрушали индексы или не давали MySQL их использовать. MySQL обычно не может применять индекс по столбцу, если этот столбец не изолирован в запросе. Изоляция столбца означает, что он не будет частью выражения или аргументом функции.

В качестве примера приведем запрос, который не может использовать индекс по столбцу `actor_id`:

```
mysql> SELECT actor_id FROM sakila.actor WHERE actor_id + 1 = 5;
```

Человеку очевидно, что выражение `WHERE` эквивалентно выражению `actor_ id = 4`. Но MySQL не умеет решать уравнения, так что это придется делать вам. Следует выработать привычку упрощать критерии во фразе `WHERE` так, чтобы индексированный столбец оказывался в одиночестве по одну сторону от оператора сравнения.

Приведем пример другой распространенной ошибки:

```
mysql> SELECT ... WHERE TO_DAYS(CURRENT_DATE) - TO_DAYS(date_col) <= 10;
```

Префиксные индексы и селективность индекса

Иногда бывает нужно проиндексировать очень длинные символьные столбцы, из-за чего индексы становятся большими и медленными. Одной из стратегий улучшения ситуации является имитация хеш-индекса, как мы показали ранее в этой главе. Но порой этого недостаточно. Что еще можно сделать?

Часто можно сэкономить пространство и добиться хорошей производительности, проиндексировав первые несколько символов, а не все значение. Индекс будет занимать меньше места, однако станет менее *селективным (избирательным)*. Селективность индекса — это отношение количества различных проиндексированных значений *(кардинальности)* к общему количеству строк в таблице (`#T`). Диапазон возможных значений селективности варьируется от 1/`#T` до 1. Индекс с высокой селективностью хорош тем, что позволяет MySQL при поиске соответствий отфильтровывать больше строк. Уникальный индекс имеет селективность, равную 1, — лучше не бывает.

Префикс столбца часто оказывается достаточно избирательным, чтобы обеспечить хорошую производительность. Индексируя столбцы типа `BLOB` или `TEXT` либо очень длинные столбцы типа `VARCHAR`, вы *обязаны* определять префиксные индексы, поскольку MySQL не позволяет индексировать такие столбцы по их полной длине.

Основная проблема заключается в выборе длины префикса, которая должна быть достаточно велика, чтобы обеспечить хорошую селективность, но не слишком велика, чтобы сэкономить место. Префикс должен быть настолько длинным, чтобы польза от его применения была почти такой же, как от использования индекса по полному столбцу. Другими словами, кардинальность префикса должна быть почти такой же, как кардинальность всего столбца.

Для определения подходящей длины префикса найдите наиболее часто встречающиеся значения и сравните их перечень со списком чаще всего используемых префиксов. В тестовой базе данных Sakila нет подходящего примера для демонстрации, поэтому создадим таблицу на основе таблицы `city`, чтобы у нас было достаточно данных для работы:

```
CREATE TABLE sakila.city_demo(city VARCHAR(50) NOT NULL);
INSERT INTO sakila.city_demo(city) SELECT city FROM sakila.city;
-- Повторить следующую команду пять раз:
INSERT INTO sakila.city_demo(city) SELECT city FROM sakila.city_demo;
-- Распределим данные в случайном порядке (неэффективно, но удобно):
UPDATE sakila.city_demo
    SET city = (SELECT city FROM  sakila.city ORDER BY RAND() LIMIT 1);
```

Теперь у нас есть тестовый набор значений. Данные распределены не очень реалистично. Кроме того, поскольку мы использовали функцию `RAND()`, ваши результаты будут отличаться от полученных нами, но для примера это непринципиально. Прежде всего найдем наиболее часто встречающиеся города:

```
mysql> SELECT COUNT(*) AS cnt, city
    -> FROM sakila.city_demo GROUP BY city ORDER BY cnt DESC LIMIT 10;
```

```
+-----+----------------+
| cnt | city           |
+-----+----------------+
|  65 | London         |
|  49 | Hiroshima      |
|  48 | Teboksary      |
|  48 | Pak Kret       |
|  48 | Yaound         |
|  47 | Tel Aviv-Jaffa |
|  47 | Shimoga        |
|  45 | Cabuyao        |
|  45 | Callao         |
|  45 | Bislig         |
+-----+----------------+
```

Обратите внимание на то, что каждое значение встречается от 45 до 65 раз. Теперь найдем наиболее часто встречающиеся *префиксы* названий городов. Начнем с трехбуквенных:

```
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 3) AS pref
    -> FROM sakila.city_demo GROUP BY pref ORDER BY cnt DESC LIMIT 10;
+-----+------+
| cnt | pref |
+-----+------+
| 483 | San  |
| 195 | Cha  |
| 177 | Tan  |
| 167 | Sou  |
| 163 | al-  |
| 163 | Sal  |
| 146 | Shi  |
| 136 | Hal  |
| 130 | Val  |
| 129 | Bat  |
+-----+------+
```

Каждый префикс встречается значительно чаще, чем город, поэтому уникальных префиксов намного меньше, чем уникальных полных названий городов. Идея состоит в увеличении длины префикса до тех пор, пока он не станет почти таким же селективным, как полная длина столбца. По результатам экспериментов мы определили, что семи символов вполне достаточно:

```
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 7) AS pref
    -> FROM sakila.city_demo GROUP BY pref ORDER BY cnt DESC LIMIT 10;
+-----+---------+
| cnt | pref    |
+-----+---------+
|  70 | Santiag |
|  68 | San Fel |
|  65 | London  |
|  61 | Valle d |
|  49 | Hiroshi |
|  48 | Teboksa |
|  48 | Pak Kre |
|  48 | Yaound  |
|  47 | Tel Avi |
|  47 | Shimoga |
+-----+---------+
```

Другой способ определить подходящую длину префикса состоит в вычислении селективности полного столбца и попытках подобрать длину префикса, обеспечивающую близкую селективность. Селективность полного столбца можно найти следующим образом:

```
mysql> SELECT COUNT(DISTINCT city)/COUNT(*) FROM sakila.city_demo;
+-------------------------------+
| COUNT(DISTINCT city)/COUNT(*) |
+-------------------------------+
|                        0.0312 |
+-------------------------------+
```

В среднем (с небольшой оговоркой) префикс будет примерно так же хорош, если его селективность около 0,031. В одном запросе можно посчитать селективность нескольких разных длин префиксов, что особенно полезно для очень больших таблиц. Это можно сделать следующим образом:

```
mysql> SELECT COUNT(DISTINCT LEFT(city, 3))/COUNT(*) AS sel3,
    ->    COUNT(DISTINCT LEFT(city, 4))/COUNT(*) AS sel4,
    ->    COUNT(DISTINCT LEFT(city, 5))/COUNT(*) AS sel5,
    ->    COUNT(DISTINCT LEFT(city, 6))/COUNT(*) AS sel6,
    ->    COUNT(DISTINCT LEFT(city, 7))/COUNT(*) AS sel7
    -> FROM sakila.city_demo;
+--------+--------+--------+--------+--------+
| sel3   | sel4   | sel5   | sel6   | sel7   |
+--------+--------+--------+--------+--------+
| 0.0239 | 0.0293 | 0.0305 | 0.0309 | 0.0310 |
+--------+--------+--------+--------+--------+
```

Этот запрос показал, что увеличение длины префикса дает небольшое улучшение селективности по мере приближения к семи символам.

Недостаточно обращать внимание только на среднюю селективность. Следует подумать (в этом и состояла оговорка) также о селективности в худшем случае. На основе средней селективности вы можете прийти к выводу, что префикса длиной четыре или пять символов достаточно, но если данные распределены очень неравномерно, это может завести вас в ловушку. Если вы посмотрите на количество вхождений наиболее распространенных префиксов названия города при использовании длины 4, то увидите явную неравномерность:

```
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 4) AS pref
    -> FROM sakila.city_demo GROUP BY pref ORDER BY cnt DESC LIMIT 5;
+-----+------+
| cnt | pref |
+-----+------+
| 205 | San  |
| 200 | Sant |
| 135 | Sout |
| 104 | Chan |
|  91 | Toul |
+-----+------+
```

При длине четыре символа наиболее распространенные префиксы встречаются значительно чаще, чем самые распространенные полные значения. То есть селективность по этим значениям ниже, чем средняя. Если у вас есть более реалистичный набор данных, чем эта сгенерированная случайным образом выборка, то,

вероятно, эффект может оказаться значительно более выраженным. Например, построение четырехзначного префиксного индекса по реальным названиям городов мира даст высокую селективность по городам, начинающимся на San и New, которых много.

Теперь, определив подходящую длину префикса для тестовых данных, создадим индекс по префиксу столбца:

```
mysql> ALTER TABLE sakila.city_demo ADD KEY (city(7));
```

Префиксные индексы могут стать хорошим способом уменьшения размера и повышения быстродействия индекса, но у них есть и недостатки: MySQL не может использовать префиксные индексы ни для запросов с фразами ORDER BY и GROUP BY, ни как покрывающие индексы.

Мы знаем еще один распространенный способ получения выигрыша от префиксных индексов — применение длинных шестнадцатеричных идентификаторов. В предыдущей главе мы обсудили эффективные приемы хранения таких идентификаторов, но что, если вы используете пакетное решение, которое невозможно изменить? Мы неоднократно встречали использование длинных шестнадцатеричных строк у vBulletin и других приложений, использующих MySQL для хранения сеансов сайта. Добавление индекса по первым восьми символам или около того часто значительно повышает производительность.

Иногда целесообразно создавать суффиксные индексы (например, для поиска всех адресов электронной почты из определенного домена). MySQL не поддерживает индексы с реверсированным ключом (https://habrahabr.ru/post/102785/), но вы можете самостоятельно хранить реверсированные строки и создавать по ним префиксный индекс. Поддерживать этот индекс можно с помощью триггеров (об этом мы говорили ранее в данной главе).

Многостолбцовые индексы

Люди часто плохо разбираются в многостолбцовых индексах. Часто ошибки состоят в индексировании многих или всех столбцов по отдельности или их индексировании в неправильном порядке.

Порядок столбцов мы обсудим в следующем разделе. У первой ошибки — индексирования множества столбцов по отдельности — есть характерная подпись в команде SHOW CREATE TABLE:

```
CREATE TABLE t (
   c1 INT,
   c2 INT,
   c3 INT,
   KEY(c1),
   KEY(c2),
   KEY(c3)
);
```

Такая стратегия индексации часто оказывается результатом того, что люди дают неопределенные, но авторитетно звучащие рекомендации, например: «Создавайте индексы в столбцах, которые появляются в условии WHERE». Этот совет ошибочен. В лучшем случае вы получите индекс с одной звездой. Такие индексы могут быть на много порядков медленнее, чем действительно оптимальные индексы. Иногда, когда вы не можете создать трехзвездочный индекс, гораздо лучше игнорировать фразу WHERE и обратить внимание на оптимальный порядок строк или создавать вместо этого покрывающий индекс.

Отдельные индексы по множеству столбцов в большинстве случаев не помогут MySQL повысить производительность запросов. MySQL 5.0 и более поздние версии могут отчасти решить проблему таких плохо проиндексированных таблиц, используя стратегию, известную как *слияние индексов*, которая позволяет запросу ограничить применение нескольких индексов в одной таблице для поиска нужных строк. Более ранние версии MySQL могли применять только один индекс, поэтому, когда ни один индекс не был достаточно хорош, MySQL часто выбирала сканирование таблицы. Например, в таблице film_actor есть индекс на film_id и индекс на actor_id, но ни один из них, будучи выбранным, не даст хорошего результата для обоих условий WHERE в этом запросе:

```
mysql> SELECT film_id, actor_id FROM sakila.film_actor
    -> WHERE actor_id = 1 OR film_id = 1;
```

Однако в MySQL 5.0 и более поздних версиях запрос может использовать оба индекса, проводя поиск по ним одновременно и объединяя результаты. Могут применяться три варианта алгоритма: объединение (условие OR), пересечение (условие AND) и объединение пересечений (комбинация обоих условий). В следующем запросе работа двух индексов объединяется, в чем вы можете убедиться, просмотрев столбец Extra:

```
mysql> EXPLAIN SELECT film_id, actor_id FROM sakila.film_actor
    -> WHERE actor_id = 1 OR film_id = 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: index_merge
possible_keys: PRIMARY,idx_fk_film_id
          key: PRIMARY,idx_fk_film_id
      key_len: 2,2
          ref: NULL
         rows: 29
        Extra: Using union(PRIMARY,idx_fk_film_id); Using where
```

MySQL может использовать эту методику для сложных запросов, поэтому для некоторых запросов вы увидите вложенные операции в столбце Extra. Стратегия слияния индексов иногда работает очень хорошо, но чаще означает, что таблица плохо проиндексирована.

- Если сервер использует пересечение индексов (обычно для условия AND), это чаще всего говорит о том, что вам нужен один индекс со всеми столбцами, а не несколько объединенных индексов.

- Если сервер использует объединение индексов (обычно для условия OR), то операции буферизации, сортировки и слияния могут задействовать большое количество ресурсов процессора и памяти. Это особенно верно, если не все индексы селективны и сканирование для операции слияния возвращает много строк.
- Напомним, что оптимизатор не учитывает эти затраты — он оптимизирует только количество произвольных чтений страницы. В итоге запрос может оказаться недооцененным, фактически его выполнение будет медленнее, чем обычное сканирование таблицы. Кроме того, интенсивное использование памяти и процессора могут влиять на конкурентные запросы, однако вы не увидите этого эффекта при изолированном выполнении запроса. Иногда переписывание таких запросов с помощью UNION, как вы привыкли делать в MySQL 4.1 и более ранних версиях, более оптимально.

Когда вы видите слияние индексов в результатах команды `EXPLAIN`, следует изучить структуру запроса и таблицы, чтобы убедиться, что их нельзя улучшить. Слияние индексов можно запретить с помощью параметра `optimizer_switch`. Кроме того, существует команда `IGNORE INDEX`.

Выбор правильного порядка столбцов

Одной из наиболее распространенных причин проблем является, как мы видели, порядок столбцов в индексе. Правильный порядок зависит от запросов, которые будут использовать индекс, поэтому стоит заранее подумать о том, как выбрать индексный порядок так, чтобы строки были отсортированы и сгруппированы наилучшим с точки зрения запроса образом. (Кстати, этот раздел относится только к индексам, упорядоченным на основе B-дерева, — хеш и другие типы индексов не сортируют свои данные.)

Порядок столбцов в индексе, упорядоченном на основе B-дерева, означает, что индекс сортируется сначала по столбцу, расположенному слева, затем по следующему и т. д. Поэтому индекс можно просматривать как в прямом, так и в обратном порядке, чтобы удовлетворить запросы с фразами `ORDER BY`, `GROUP BY` и `DISTINCT`, в которых порядок столбцов точно соответствует индексу.

Таким образом, порядок столбцов жизненно важен в многостолбцовых индексах. Он позволяет индексу зарабатывать звезды по трехзвездочной системе Лахденмаки и Лича (о ней говорилось в разделе «Преимущества индексов» ранее в этой главе). Далее мы приведем много примеров того, как это работает.

Порядок столбцов можно определить с помощью старого эмпирического правила, согласно которому первым делом нужно поместить в индекс столбцы с наибольшей селективностью. Полезна ли эта рекомендация? В некоторых случаях она может быть весьма полезна, но обычно гораздо важнее избегать произвольных операций ввода/вывода и сортировок. (Конкретные ситуации отличаются друг от друга, поэтому не может быть правила на все случаи жизни. Так что это эмпирическое правило, скорее всего, не так полезно, как вы думаете.)

Помещать в индекс сначала наиболее селективные столбцы может быть целесообразно тогда, когда целью его создания является оптимизация поиска WHERE, а сортировка и группировка не принимаются во внимание. В таких случаях было бы неплохо разработать индекс, который максимально быстро отфильтровывал бы строки и поэтому лучше всего подходил бы для запросов, в которых указывается только префикс индекса в разделе WHERE. Однако это зависит не только от селективности (общей кардинальности) столбцов, но и от фактических значений, используемых для поиска строк, — распределения значений. Тем же принципом мы пользовались, когда выбирали оптимальную длину префикса. По-видимому, действительно нужно выбрать такой порядок столбцов, который будет максимально селективным для наиболее часто встречающихся запросов.

Рассмотрим в качестве примера следующий запрос:

```
SELECT * FROM payment WHERE staff_id = 2 AND customer_id = 584
```

Следует ли создать индекс (staff_id, customer_id) или стоит поменять порядок столбцов на обратный? Мы можем запустить несколько быстрых запросов, чтобы понять, как распределяются значения в таблице, и определить, какой столбец имеет более высокую селективность. Преобразуем запрос для подсчета кардинальности каждого предиката[1] в запросе с фразой WHERE:

```
mysql> SELECT SUM(staff_id = 2), SUM(customer_id = 584) FROM payment\G
*************************** 1. row ***************************
SUM(staff_id = 2): 7992
SUM(customer_id = 584): 30
```

Согласно эмпирическому правилу следует сначала поместить в индекс customer_id, потому что предикат соответствует меньшему количеству строк в таблице. Затем мы снова запустим запрос, чтобы узнать, насколько селективен staff_id в пределах диапазона строк, выбранных для клиента с данным идентификатором:

```
mysql> SELECT SUM(staff_id = 2) FROM payment WHERE customer_id = 584\G
*************************** 1. row ***************************
SUM(staff_id = 2): 17
```

Применяйте эту методику с осторожностью, поскольку результаты зависят от конкретных констант, заданных для выбранного запроса. Если вы оптимизируете свои индексы для этого запроса, но не для других, производительность сервера в целом может ухудшиться, а некоторые запросы могут выполняться непредсказуемо.

Если вы используете худшую выборку запроса из отчета какого-либо инструмента, например pt-query-digest, эта методика может помочь определить наиболее полезные индексы для ваших запросов и данных. Но если у вас нет конкретных образцов для запуска, стоит опираться на старое эмпирическое правило: проверять кардинальность не по одному запросу:

```
mysql> SELECT COUNT(DISTINCT staff_id)/COUNT(*) AS staff_id_selectivity,
    > COUNT(DISTINCT customer_id)/COUNT(*) AS customer_id_selectivity,
```

[1] Гики оптимизации называют его sarg, от searchable argument — «доступный для поиска параметр». Теперь вы тоже гик!

```
    > COUNT(*)
    > FROM payment\G
*************************** 1. row ***************************
   staff_id_selectivity: 0.0001
customer_id_selectivity: 0.0373
               COUNT(*): 16049
```

У столбца `customer_id` самая высокая селективность, поэтому можно рекомендовать поставить этот столбец первым в индексе:

```
mysql> ALTER TABLE payment ADD KEY(customer_id, staff_id);
```

Как и в случае с префиксными индексами, проблемы часто возникают из-за особых значений, кардинальность которых превышает нормальную. Например, нам доводилось встречать приложения, которые квалифицировали незалогированных пользователей как гостей. Такие пользователи получали специальный идентификатор пользователя в таблице сеанса и других местах, где фиксировалась пользовательская активность. Запросы, связанные с этим идентификатором пользователя, скорее всего, будут отличаться от других запросов, поскольку, как правило, существует множество сеансов, пользователи которых не залогинились. Мы также не раз видели, как системные аккаунты вызывают аналогичные проблемы. У одного приложения был волшебный аккаунт администратора: она был не настоящим пользователем, а «другом» всех пользователей сайта. С этого аккаунта могли отправляться уведомления о статусе и другие сообщения. Огромный список «друзей» этого пользователя вызывал серьезные проблемы с производительностью на сайте.

В реальности такое встречается довольно часто. Любое отклонение от стандартной ситуации, даже если это не результат применения плохих решений по проектированию и управлению приложением, может вызвать проблемы. Пользователи, у которых действительно есть много друзей, фотографий, сообщений о статусе и т. п., могут создавать такие же проблемы, как и боты.

Приведем пример, с которым мы однажды столкнулись на форуме, где пользователи обменивались историями и впечатлениями о продукте. Запросы этой конкретной формы выполнялись очень медленно:

```
mysql> SELECT COUNT(DISTINCT threadId) AS COUNT_VALUE
    -> FROM Message
    -> WHERE (groupId = 10137) AND (userId = 1288826) AND (anonymous = 0)
    -> ORDER BY priority DESC, modifiedDate DESC
```

Кажется, что у этого запроса не слишком удачный индекс, поэтому клиент попросил нас посмотреть, можно ли его улучшить. Команда `EXPLAIN` выдала следующую информацию:

```
         id: 1
select_type: SIMPLE
      table: Message
       type: ref
        key: ix_groupId_userId
    key_len: 18
        ref: const,const
       rows: 1251162
      Extra: Using where
```

Индекс, который MySQL применила для этого запроса по столбцам (`groupId, userId`), кажется хорошим выбором при отсутствии информации о кардинальности столбцов. Однако получится совсем другая картина, если мы посмотрим, сколько строк соответствует идентификатору пользователя и идентификатору группы:

```
mysql> SELECT COUNT(*), SUM(groupId = 10137),
    -> SUM(userId = 1288826), SUM(anonymous = 0)
    -> FROM Message\G
*************************** 1. row ***************************
            count(*): 4142217
 sum(groupId = 10137): 4092654
sum(userId = 1288826): 1288496
   sum(anonymous = 0): 4141934
```

Оказалось, что этой группе соответствовала почти каждая строка в таблице, а пользователю соответствовали 1,3 миллиона строк — в этом случае подходящего индекса просто не существует! Это вызвано тем, что данные были перенесены из другого приложения и все сообщения приписаны пользователю с правами администратора и группе как части процесса импорта. Решение данной проблемы состояло в изменении кода приложения таким образом, чтобы распознавать этого специального пользователя и его группу и не выполнять для него запросов.

Мораль этой маленькой истории состоит в том, что эмпирические и эвристические правила могут оказаться полезными, но вы должны быть внимательными и не рассчитывать на то, что усредненная производительность соответствует производительности любого случая, включая особые. Такие случаи могут ухудшить производительность всего приложения.

Напоследок добавим, что хотя интересно проверять эмпирическое правило о селективности и кардинальности, другие факторы, такие как сортировка, группировка и наличие условий диапазона в разделе `WHERE`, могут намного сильнее изменить производительность запросов.

Кластерные индексы

Кластерные индексы[1] — это не отдельный тип индекса, а скорее, подход к хранению данных. Детали могут различаться в зависимости от реализации, но в InnoDB кластерный индекс фактически содержит в одной и той же структуре и индекс, упорядоченный на основе B-дерева, и сами строки.

Когда над таблицей построен кластерный индекс, ее строки хранятся в листьях индекса. Термин «кластерный» означает, что строки с близкими значениями ключа хранятся рядом друг с другом[2]. Над таблицей можно построить только один кластерный индекс, поскольку невозможно хранить одну и ту же строку одновременно

[1] Пользователи Oracle наверняка знакомы с термином «индексно-организованная таблица», который означает то же самое.

[2] Это не всегда так, и вы скоро узнаете почему.

в двух местах (*покрывающие индексы* позволяют эмулировать несколько кластерных индексов, но об этом поговорим позднее).

За реализацию индексов отвечают подсистемы хранения, однако не все они поддерживают кластерные индексы. В данном разделе будем говорить исключительно об InnoDB, но обсуждаемые принципы, по крайней мере частично, применимы к любой подсистеме хранения, которая поддерживает кластерные индексы сейчас или станет поддерживать их в будущем.

На рис. 5.3 показано, как располагаются записи в кластерном индексе. Обратите внимание на то, что листья содержат сами строки, а узлы — только индексированные столбцы. В рассматриваемом примере индексированный столбец содержит целочисленные значения.

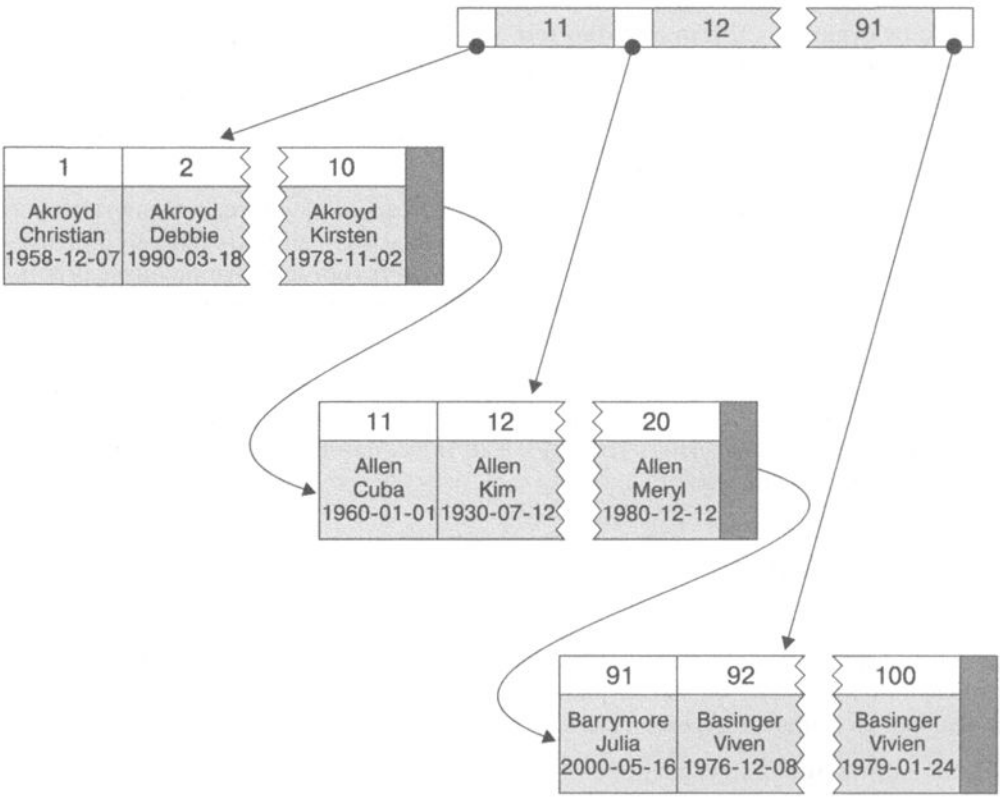


Рис. 5.3. Расположение записей в кластерном индексе

Некоторые серверы базы данных позволяют выбрать, какой индекс сделать кластерным, но на момент написания книги ни одна из встроенных подсистем хранения MySQL не предоставляла такой возможности. InnoDB кластеризует данные по первичному ключу. Это означает, что индексированный столбец на рис. 5.3 является столбцом, содержащим первичный ключ.

Если вы не определили первичный ключ, то InnoDB попытается использовать вместо него уникальный индекс, не допускающий значений `NULL`. Если такого индекса не существует, InnoDB определит скрытый первичный ключ за вас и затем по нему выполнит кластеризацию таблицы. InnoDB размещает записи вместе только внутри страницы. Разные страницы с близкими значениями ключей могут оказаться далеко друг от друга.

Как правило, первичный кластерный ключ увеличивает производительность, но иногда он может вызвать серьезные проблемы с ней. Таким образом, решение о кластеризации нужно тщательно обдумывать, особенно при замене подсистемы хранения с InnoDB на какую-то другую и наоборот.

Кластеризованные данные имеют несколько очень важных достоинств.

- Возможность хранить связанные данные рядом. Например, при реализации почтового ящика можете выполнить кластеризацию таблицы по столбцу `user_id`, тем самым для выборки всех сообщений одного пользователя нужно будет прочитать с диска лишь небольшое количество страниц. Если не использовать кластеризацию, то для каждого сообщения может потребоваться отдельная операция дискового ввода/вывода.
- Быстрый доступ к данным. Кластерный индекс хранит и индекс, и данные вместе в одном B-дереве, поэтому извлечение строк из кластерного индекса обычно происходит быстрее, чем сопоставимый поиск в некластерном индексе.
- Запросы, которые применяют покрывающие индексы, могут использовать значение первичного ключа из листа.

Эти преимущества могут значительно увеличить производительность, если вы спроектируете свои таблицы и запросы с их учетом. Однако у кластерных индексов есть и недостатки.

- Кластеризацию выгодно использовать, когда рабочая нагрузка характеризуется большим количеством операций ввода/вывода. Если данные помещаются в памяти и порядок доступа к ним не имеет значения, то кластерные индексы не принесут большой пользы.
- Скорость операций вставки сильно зависит от ее порядка. Вставка строк в порядке, соответствующем первичному ключу, является самым быстрым способом загрузить данные в таблицу InnoDB. Если вы загружаете большое количество данных в другом порядке, то по окончании загрузки целесообразно реорганизовать таблицу командой `OPTIMIZE TABLE`.
- Обновление столбцов кластерного индекса весьма затратно, поскольку InnoDB вынуждена перемещать каждую обновленную строку на новое место.
- Для таблиц с кластерным индексом вставка новых строк или обновление первичного ключа, требующее перемещения строки, могут привести к разделению страницы. Это происходит тогда, когда значение ключа строки таково, что строка должна быть помещена в страницу, заполненную данными. Чтобы строка поместилась, подсистема хранения вынуждена разделить страницу на две. Из-за этого таблица может занять больше места на диске.

- Полное сканирование кластерных таблиц может оказаться медленным, особенно если строки упакованы менее плотно или из-за разделения страниц хранятся неупорядоченно.
- Вторичные (некластерные) индексы могут оказаться больше, чем вы ожидаете, поскольку в их листьях хранятся значения столбцов, составляющих первичный ключ.
- Для получения доступа к данным по вторичному индексу требуется просматривать два индекса вместо одного.

Последний момент может быть не совсем ясен. Почему доступ с использованием вторичного индекса требует двух операций просмотра? Ответ заключается в природе указателей на строки, которые хранятся во вторичном индексе. Не забывайте, что лист содержит не указатель на физический адрес строки, а значение ее первичного ключа. Это означает, что для нахождения строки по вторичному индексу подсистема хранения сначала ищет в нем лист, а затем использует содержащееся в нем значение первичного ключа для поиска самой строки. Это двойная работа — два прохода по B-дереву вместо одного[1]. В InnoDB адаптивный хеш-индекс помогает уменьшить эти потери.

Сравнение размещения данных в InnoDB и MyISAM

Различия в организации кластеризованного и некластеризованного размещения данных и связанная с этим разница между первичными и вторичными индексами могут привести к путанице и возникновению неожиданностей. Давайте рассмотрим, как InnoDB и MyISAM разместят данные следующей таблицы:

```
CREATE TABLE layout_test (
    col1 int NOT NULL,
    col2 int NOT NULL,
    PRIMARY KEY(col1),
    KEY(col2)
);
```

Предположим, в таблицу в случайном порядке были добавлены 10 000 строк с первичными ключами в диапазоне от 1 до 10 000. Затем выполнена оптимизация с помощью команды `OPTIMIZE TABLE`. Другими словами, данные размещены на диске оптимальным образом, но строки могут располагаться в случайном порядке. Элементам столбца `col2` присвоены случайные значения между 1 и 100, поэтому в таблице множество дубликатов.

Размещение данных в MyISAM. Размещение данных в подсистеме MyISAM проще, поэтому покажем его первым. MyISAM сохраняет строки на диске в том порядке, в котором они были вставлены, что показано на рис. 5.4.

---

[1] Кстати, некластеризованные индексы не всегда могут обеспечить поиск строк в один заход. Когда строка изменяется, она больше не помещается на исходное место, поэтому может столкнуться в таблице с фрагментированными строками или переадресацией, что приведет к увеличению работы по поиску строки.

Номер ряда	col1	col2
0	99	8
1	12	56
2	3000	62
...	...	...
9997	18	8
9998	4700	13
9999	3	93

Рис. 5.4. Размещение данных для таблицы layout_test в MyISAM

Рядом со строками мы привели их номера, начиная с нуля. Поскольку строки имеют фиксированный размер, MyISAM может найти любую из них, смещаясь на нужное количество байтов от начала таблицы (MyISAM не всегда использует номера строк, которые мы показали, эта подсистема хранения применяет различные стратегии в зависимости от того, имеют строки фиксированный или переменный размер).

При таком размещении построение индекса не вызывает сложностей. Мы проиллюстрируем это с помощью последовательности диаграмм, отбросив такие физические детали, как страницы, и показывая только узлы индекса. Каждый лист в индексе просто содержит номер строки. На рис. 5.5 показан первичный ключ таблицы.

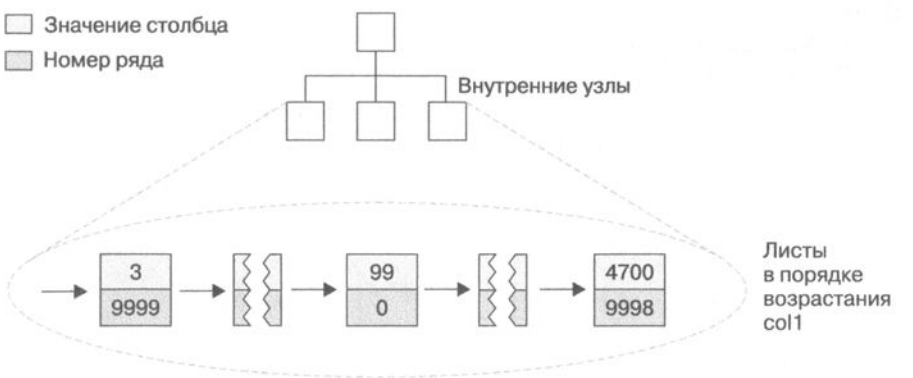


Рис. 5.5. Размещение первичного ключа для таблицы layout_test в MyISAM

Мы опустили некоторые детали, например то, сколько внутренних узлов-потомков может быть у узла B-дерева, поскольку это неважно для общего понимания того, как размещаются данные в некластерной подсистеме хранения.

Что можно сказать об индексе по столбцу `col2`? Есть ли в нем что-то особенное? Оказывается, ничего — это такой же индекс, как любой другой. На рис. 5.6 показан индекс по столбцу `col2`.

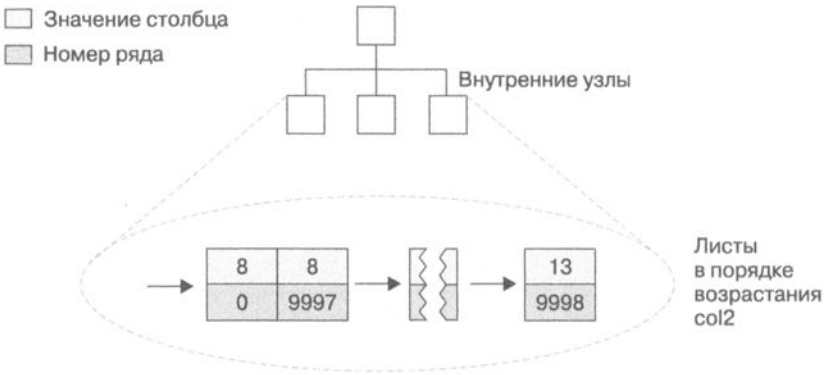


Рис. 5.6. Размещение индекса по столбцу col2 для таблицы layout_test в MyISAM

Фактически в MyISAM нет никаких структурных различий между первичным ключом и любым другим индексом. Первичный ключ является просто уникальным, не допускающим значений `NULL` индексом под названием `PRIMARY`.

Размещение данных в InnoDB. InnoDB хранит те же самые данные совсем по-другому из-за своей кластерной организации. InnoDB хранит таблицу так, как показано на рис. 5.7.

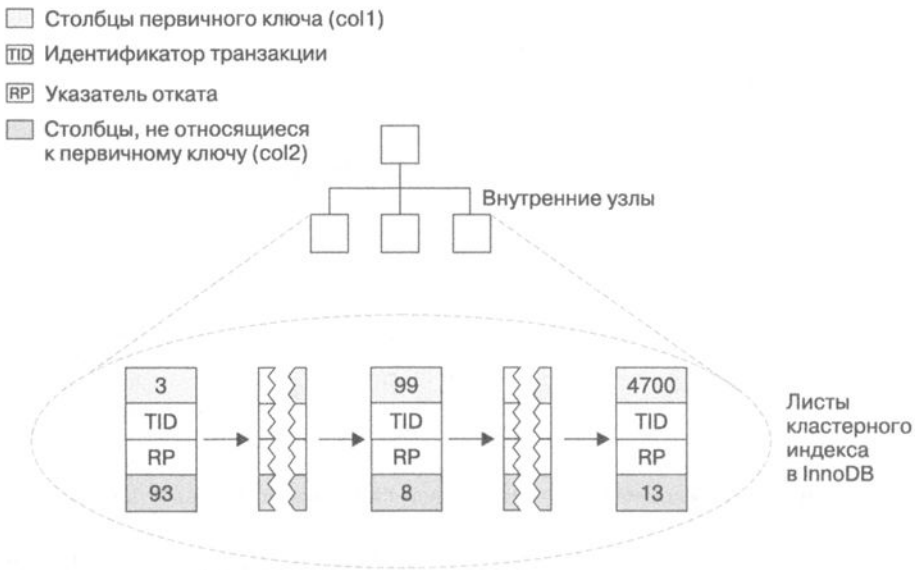


Рис. 5.7. Размещение первичного ключа для таблицы layout_test в InnoDB

На первый взгляд, здесь особых отличий от рис. 5.5 нет. Но если посмотреть внимательнее, можно заметить, что показана вся таблица, а не только индекс. Поскольку

кластерный индекс в InnoDB является таблицей, то отдельного хранилища для строк, как в MyISAM, нет.

Каждый лист в кластерном индексе содержит значение первичного ключа, идентификатор транзакции и указатель отката, который InnoDB использует для поддержки транзакций и механизма MVCC, а также прочие столбцы (в данном случае `col2`). Если первичный ключ создан по префиксу столбца, то полное значение этого столбца хранится вместе с прочими столбцами.

Кроме того, в отличие от MyISAM, в InnoDB вторичные индексы совсем не такие, как кластерные. Вместо хранения указателей на строки листья вторичных индексов содержат значения первичного ключа, которые служат такими указателями на строку. Подобная стратегия уменьшает объем работы, необходимой для поддержки вторичных индексов при перемещении строки или для разделения страницы данных. Использование значений первичного ключа строки в качестве указателя увеличивает размер индекса и означает, что InnoDB может переместить строку без обновления указателей на нее.

Рисунок 5.8 иллюстрирует индекс по столбцу `col2` для таблицы из примера. Каждый лист содержит индексированные столбцы (в данном случае только `col2`), за которыми следуют значения первичного ключа (`col1`).

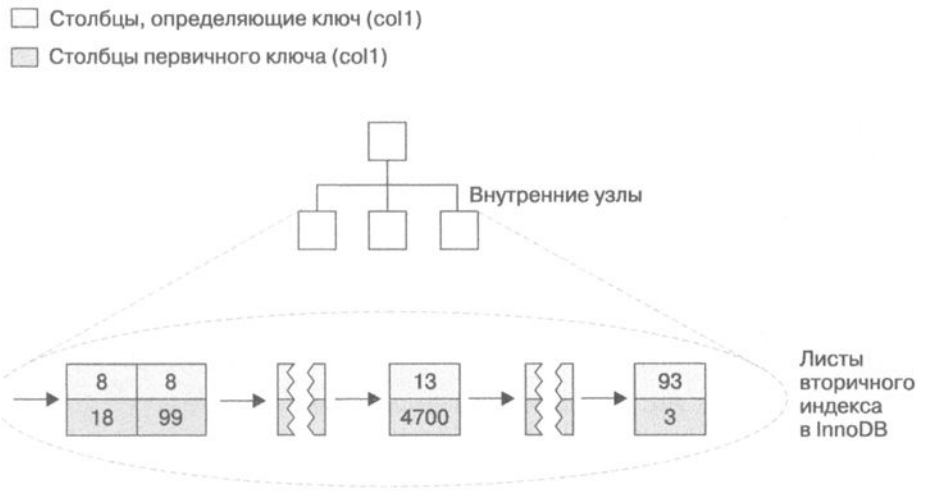


Рис. 5.8. Размещение вторичного индекса для таблицы layout_test в InnoDB

Здесь показаны листья B-дерева, однако мы умышленно опустили детали, касающиеся нелистовых узлов. Каждый нелистовой узел B-дерева в InnoDB содержит индексированный столбец (-цы) и указатель на узел следующего уровня (которым может быть либо другой нелистовой узел, либо лист). Это относится ко всем индексам, как кластерным, так и вторичным.

На рис. 5.9 показано абстрактное представление организации таблицы в InnoDB и MyISAM. Легко увидеть различия в том, как хранятся данные и индексы в этих двух системах.

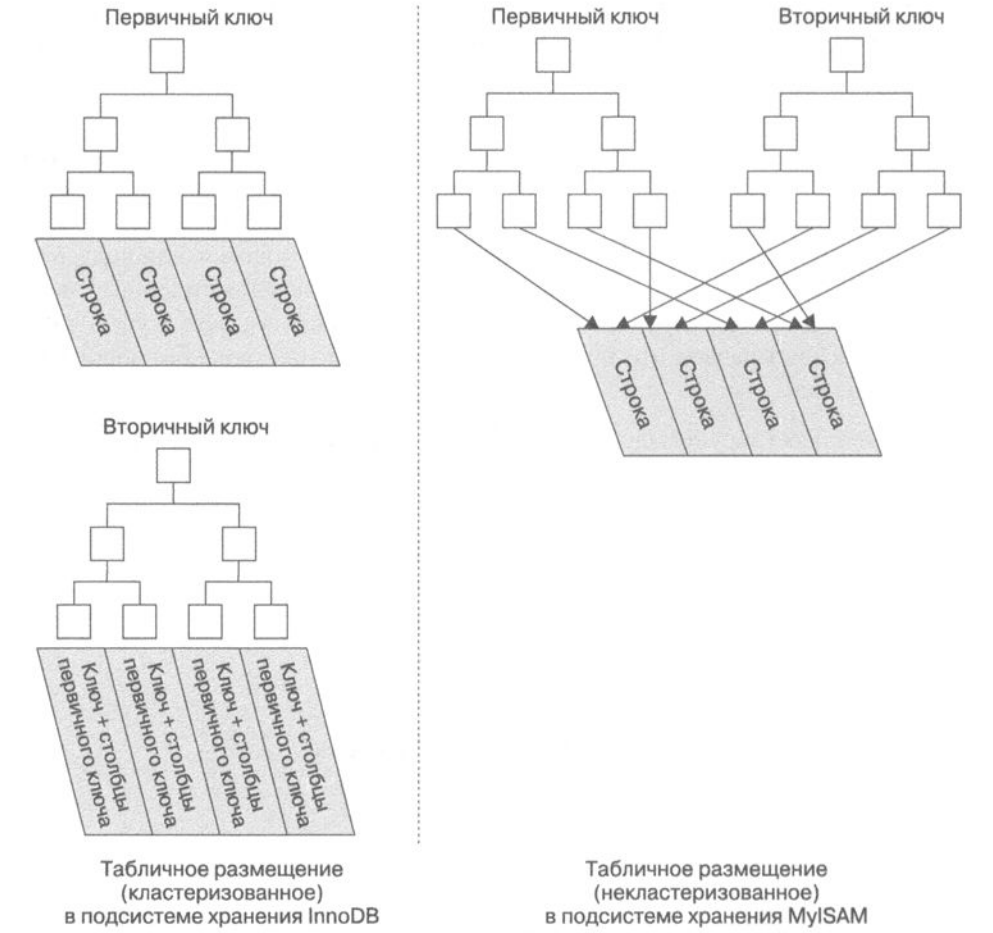


Рис. 5.9. Кластерные и некластерные таблицы

Если вы не понимаете, чем различаются кластерное и некластерное хранение и почему это так важно, не беспокойтесь. Все станет яснее, когда вы изучите больше материала, особенно в этой и следующей главах. Данные концепции довольно сложны, и для того, чтобы в них хорошо разобраться, требуется время.

Вставка строк в порядке первичного ключа в InnoDB

Если вы используете InnoDB и вам не требуется особая кластеризация, то целесообразно определить *суррогатный ключ*, то есть первичный ключ, значение которого не вытекает из данных вашего приложения. Проще всего воспользоваться для этого

столбцом с атрибутом AUTO_INCREMENT. Это гарантирует, что значение поля, по которому построен первичный ключ, монотонно возрастает, тем самым обеспечивается лучшая производительность соединений, использующих первичные ключи.

Стоит избегать случайных (непоследовательных и распределенных среди большого набора значений) кластерных ключей, особенно при большой нагрузке на систему ввода/вывода. Например, использование значений UUID — это плохой выбор с точки зрения производительности: вставка в кластерный индекс будет случайной, что является худшим сценарием и не обеспечивает полезной кластеризации данных.

Для того чтобы продемонстрировать это, мы выполнили эталонное тестирование производительности для двух ситуаций. В первом случае выполнялась вставка в таблицу userinfo с целочисленным идентификатором. Таблица была определена следующим образом:

```
CREATE TABLE userinfo (
    id              int unsigned NOT NULL AUTO_INCREMENT,
    name            varchar(64) NOT NULL DEFAULT '',
    email           varchar(64) NOT NULL DEFAULT '',
    password        varchar(64) NOT NULL DEFAULT '',
    dob             date DEFAULT NULL,
    address         varchar(255) NOT NULL DEFAULT '',
    city            varchar(64) NOT NULL DEFAULT '',
    state_id        tinyint unsigned NOT NULL DEFAULT '0',
    zip             varchar(8) NOT NULL DEFAULT '',
    country_id      smallint unsigned NOT NULL DEFAULT '0',
    gender          ('M','F')NOT NULL DEFAULT 'M',
    account_type    varchar(32) NOT NULL DEFAULT '',
    verified        tinyint NOT NULL DEFAULT '0',
    allow_mail      tinyint unsigned NOT NULL DEFAULT '0',
    parrent_account int unsigned NOT NULL DEFAULT '0',
    closest_airport varchar(3) NOT NULL DEFAULT '',
    PRIMARY KEY (id),
    UNIQUE  KEY email (email),
    KEY     country_id (country_id),
    KEY     state_id (state_id),
    KEY     state_id_2 (state_id,city,address)
) ENGINE=InnoDB
```

Обратите внимание на целочисленный автоинкрементный первичный ключ[1].

Во втором случае рассмотрим таблицу userinfo_uuid. Она практически идентична таблице userinfo, за исключением того, что первичным ключом является UUID, а не целое число:

```
CREATE TABLE userinfo_uuid (
    uuid varchar(36) NOT NULL,
    ...
```

[1] Стоит отметить, что это настоящая таблица с вторичными индексами и множеством столбцов. Если мы их удалим и проведем эталонное тестирование производительности первичного ключа, разница будет еще больше.

Мы выполнили эталонное тестирование обеих таблиц. Сначала вставили в каждую по 1 миллиону записей на сервере, имеющем достаточно памяти для размещения в ней индексов. Затем вставили в те же таблицы по 3 миллиона строк — индексы увеличились настолько, что перестали помещаться в памяти. В табл. 5.1 сравниваются результаты эталонного тестирования.

Таблица 5.1. Эталонное тестирование вставки строк в таблицы InnoDB

Таблица	Количество строк	Время, с	Размер индекса, Мбайт
userinfo	1 000 000	137	342
userinfo_uuid	1 000 000	180	544
userinfo	3 000 000	1233	1036
userinfo_uuid	3 000 000	4525	1707

Обратите внимание на то, что при использовании первичного ключа UUID не только вставка строк заняла больше времени, но и размер индекса слегка увеличился. Одна из причин — больший размер первичного ключа. Но, несомненно, влияние оказали также разделение страниц и неизбежная фрагментация.

Чтобы понять, почему возникает такая ситуация, давайте посмотрим, что происходило в индексе, когда мы вставляли данные в первую таблицу. На рис. 5.10 показано, как вставляемые строки сначала заполняют одну страницу, а затем переходят на следующую.

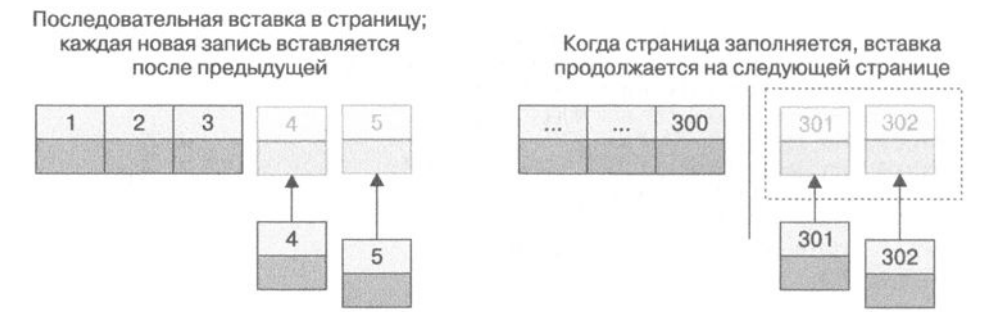


Рис. 5.10. Вставка последовательных значений индекса в кластерный индекс

Как показано на рис. 5.10, InnoDB сохраняет новую запись непосредственно после предыдущей, поскольку значения первичного ключа являются последовательными. Когда коэффициент заполнения страницы достигает максимально допустимого значения (в InnoDB коэффициент первоначального заполнения составляет 15/16, чтобы оставалось место для будущих модификаций), следующая запись размещается на новой странице. После окончания последовательной загрузки данных страницы первичных ключей оказались почти заполненными упорядоченными записями, что крайне желательно.

Сопоставьте этот процесс с тем, что происходило, когда мы вставляли данные во вторую таблицу с кластерным индексом по столбцу, содержащему UUID. Это проиллюстрировано на рис. 5.11.

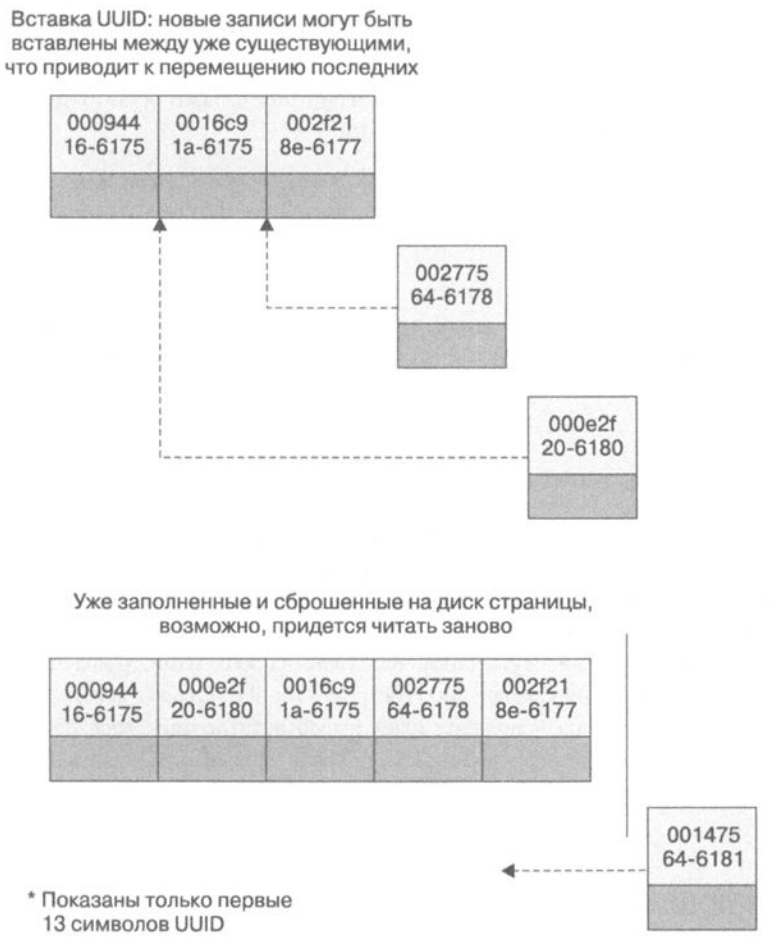


Рис. 5.11. Вставка непоследовательных значений в кластерный индекс

Поскольку значение первичного ключа в каждой последующей строке не обязательно больше, чем в предыдущей, InnoDB не всегда может разместить новую строку в конце индекса. Ей приходится искать для нее подходящее положение — обычно где-то посередине уже существующих данных — и освобождать место. Это обусловливает значительную дополнительную работу и неоптимальное размещение данных. Приведем недостатки такой ситуации.

- Страница, на которую должна попасть строка, может оказаться сброшенной на диск и удаленной из кэша или еще не размещенной в кэше. InnoDB придется

искать ее и считывать с диска, прежде чем вставить новую строку. Из-за этого приходится выполнять много произвольных операций ввода/вывода.

- Когда операции вставки осуществляются не по порядку, InnoDB часто приходится разделять страницы, чтобы освободить место для новых строк. Это требует перемещения большого объема данных и изменения по меньшей мере трех страниц вместо одной.
- Из-за разделения страницы оказываются заполненными беспорядочно и неплотно, что нередко приводит к фрагментации.

После загрузки таких случайных значений в кластерный индекс следовало бы запустить команду `OPTIMIZE TABLE`, которая перестроит таблицу и заполнит страницы оптимальным образом.

Моралью всей этой истории является то, что при использовании InnoDB нужно стремиться вставлять данные в порядке, соответствующем первичному ключу, и стараться использовать такой кластерный ключ, который монотонно возрастает для новых строк.

Когда вставка в порядке первичного ключа — это плохо

В InnoDB при рабочих нагрузках с высокой степенью параллелизма вставка в порядке первичного ключа может создавать точки конкуренции. Горячей точкой является последняя страница первичного индекса. Поскольку все вставки происходят именно здесь, возникает состязание за блокировку следующего ключа. Другой горячей точкой является блокировка. Если вы сталкиваетесь с такими проблемами, то можете перепроектировать таблицу или приложение либо настроить InnoDB (`innodb_autoinc_lock_mode`). Если ваша версия сервера не поддерживает `innodb_autoinc_lock_mode`, целесообразно перейти на более новую версию InnoDB, которая при этой рабочей нагрузке будет работать лучше.

Покрывающие индексы

Обычно индексы рекомендуется создавать для раздела `WHERE` запроса. Но они полезны для всего запроса, а не только для этого раздела. Индексы обеспечивают эффективный поиск строк, но MySQL может использовать их также для того, чтобы извлечь данные столбца, не считывая строку таблицы. В конце концов, листья индекса содержат те значения, которые они индексируют, зачем же просматривать саму строку, если чтение индекса уже может дать нужные данные? Индекс, который содержит (или *покрывает*) все данные, необходимые для формирования результатов запроса, называется *покрывающим индексом*.

Покрывающие индексы могут стать очень мощным инструментом и значительно увеличить производительность. Рассмотрим преимущества считывания индекса вместо самих данных.

- Записи индекса обычно компактнее полной строки, поэтому, читая только индексы, MySQL может обращаться к существенно меньшему объему данных. Это очень важно при кэшированной рабочей нагрузке, когда время отклика определяется в основном копированием данных. Это полезно и при большом количестве операций ввода/вывода, поскольку индексы меньше, чем данные, и лучше помещаются в памяти (что особенно справедливо в отношении MyISAM, которая может упаковывать индексы, тем самым делая их еще меньше).
- Индексы отсортированы по индексируемым значениям (по крайней мере внутри страницы), поэтому для поиска по диапазону, характеризующегося большим объемом ввода/вывода, потребуется меньше операций обращения к диску по сравнению с извлечением каждой строки из произвольного места хранения. Для некоторых подсистем хранения, например MyISAM и Percona XtraDB, вы можете даже оптимизировать таблицу командой `OPTIMIZE` и получить полностью отсортированные индексы, в результате чего для простых запросов по диапазону доступ к индексу будет вообще последовательным.
- Некоторые подсистемы хранения, такие как MyISAM, кэшируют в памяти MySQL только индексы. Поскольку кэширование данных для MyISAM выполняет операционная система, доступ к ним обычно требует системного вызова. Это может сильно повлиять на производительность, особенно при кэшированной рабочей нагрузке, когда системный вызов оказывается самой затратной частью доступа к данным.
- Покрывающие индексы особенно полезны для таблиц InnoDB. Вторичные индексы InnoDB хранят значения первичного ключа строки в листьях. Таким образом, вторичный индекс, покрывающий запрос, позволяет избежать еще одного поиска по первичному индексу.

Во всех этих сценариях выполнение запроса с использованием индекса обычно становятся значительно менее затратным, чем извлечение всей записи из таблицы.

Не каждый тип индекса может выступать в роли покрывающего. Индекс должен хранить значения индексируемых столбцов. Хеш-индексы, пространственные индексы и полнотекстовые индексы такие значения не хранят, поэтому MySQL может использовать в качестве покрывающих только индексы, упорядоченные на основе B-дерева. Кроме того, различные подсистемы хранения по-разному реализуют покрывающие индексы, а некоторые не поддерживают их вовсе (на момент написания книги такой была подсистема Memory).

Запустив команду `EXPLAIN` для запроса, который покрывается индексом (он так и называется — *покрываемый индексом запрос*), вы увидите в столбце `Extra` сообщение `Using index`[1]. Например, таблица `sakila.inventory` имеет многостолбцовый индекс

[1] Легко спутать сообщение Using index в столбце Extra с сообщением index в столбце type. Однако это разные сообщения. Столбец type не имеет никакого отношения к покрывающим индексам — он показывает тип доступа запроса или поясняет, как запрос будет находить строки. Руководство пользователя по MySQL называет его типом соединения.

по (store_id, film_id). MySQL может использовать его для выполнения запросов только по этим двум столбцам, например:

```
mysql> EXPLAIN SELECT store_id, film_id FROM sakila.inventory\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: inventory
         type: index
possible_keys: NULL
          key: idx_store_id_film_id
      key_len: 3
          ref: NULL
         rows: 4673
        Extra: Using index
```

Покрываемые индексом запросы имеют возможность отключить эту оптимизацию. Оптимизатор MySQL перед выполнением запроса принимает решение, покрывает ли его какой-либо индекс. Предположим, индекс покрывает условие WHERE, но не весь запрос. Даже если условие WHERE не выполняется, MySQL 5.5 и более ранние версии в любом случае извлекут строку, несмотря на то что она не нужна и впоследствии будет отфильтрована.

Посмотрим, почему это может произойти и как переписать запрос, чтобы обойти проблему. Начнем со следующего запроса:

```
mysql> EXPLAIN SELECT * FROM products WHERE actor='SEAN CARREY'
    -> AND title like '%APOLLO%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: products
         type: ref
possible_keys: ACTOR,IX_PROD_ACTOR
          key: ACTOR
      key_len: 52
          ref: const
         rows: 10
        Extra: Using where
```

Индекс не может покрыть этот запрос по двум причинам.

- Мы выбрали все столбцы из таблицы, а ни один индекс не покрывает все столбцы. Однако есть обходной маневр, который MySQL теоретически может использовать: в условии WHERE упоминаются только столбцы, которые покрываются индексом, поэтому MySQL может с помощью индекса найти актор, проверить, соответствует ли название заданному критерию, и только потом считать всю строку.
- MySQL не может выполнять операцию LIKE в индексе. Это ограничение API подсистемы хранения, которая в MySQL 5.5 и более ранних версиях допускает в операциях с индексами только простые сравнения (равенство, неравенство и «больше»). MySQL может выполнять сравнения LIKE по префиксу, поскольку допускает преобразование их в простые сравнения, однако наличие символа под-

становки в начале шаблона не позволяет подсистеме хранения провести сопоставление. Таким образом, самому серверу MySQL придется выполнять извлечение и сравнение значений из строки, а не из индекса.

Существует способ обойти обе проблемы, скомбинировав разумное индексирование и переписывание запроса. Мы можем расширить индекс так, чтобы он покрывал столбцы (`artist, title, prod_id`), и переписать запрос следующим образом:

```
mysql> EXPLAIN SELECT *
    -> FROM products
    ->    JOIN (
    ->       SELECT prod_id
    ->       FROM products
    ->       WHERE actor='SEAN CARREY' AND title LIKE '%APOLLO%'
    ->    ) AS t1 ON (t1.prod_id=products.prod_id)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: <derived2>
              ...опущено...
*************************** 2. row ***************************
           id: 1
  select_type: PRIMARY
        table: products
              ... опущено...
*************************** 3. row ***************************
           id: 2
  select_type: DERIVED
        table: products
         type: ref
possible_keys: ACTOR,ACTOR_2,IX_PROD_ACTOR
          key: ACTOR_2
      key_len: 52
          ref:
         rows: 11
        Extra: Using where; Using index
```

Назовем этот способ отложенным соединением, поскольку доступ к столбцам откладывается. Теперь MySQL использует покрывающий индекс на первом этапе запроса, когда ищет строки в подзапросе в разделе `FROM`. Он не использует индекс для покрытия всего запроса, но это лучше чем ничего.

Эффективность такой оптимизации зависит от того, сколько строк отвечает условию `WHERE`. Предположим, таблица `products` содержит 1 миллион строк. Посмотрим, как эти два запроса выполняются с тремя различными наборами данных, каждый из которых содержит 1 миллион строк.

1. В первом наборе данных есть 30 000 фильмов, в которых играл актер Шон Керри, в названиях 20 000 из них содержится слово Apollo.
2. Во втором наборе данных есть 30 000 фильмов, в которых играл актер Шон Керри, в названиях 40 из них содержится слово Apollo.
3. В третьем наборе данных есть 50 фильмов, в которых играл актер Шон Керри, в названиях десяти из них содержится слово Apollo.

На этих трех наборах данных мы провели эталонное тестирование двух вариантов запроса и получили результаты, показанные в табл. 5.2.

Таблица 5.2. Результаты эталонного тестирования для запросов, покрываемых и не покрываемых индексами

Набор данных	Оригинальный запрос	Оптимизированный запрос
Пример 1	5 запросов в секунду	5 запросов в секунду
Пример 2	7 запросов в секунду	35 запросов в секунду
Пример 3	2400 запросов в секунду	2000 запросов в секунду

Интерпретировать эти результаты нужно следующим образом.

- В примере 1 запрос возвращает большой результирующий набор, поэтому мы не видим эффекта от оптимизации. Большая часть времени потрачена на чтение и отправку данных.
- В примере 2, где второе условие фильтрации оставляет небольшой набор результатов после фильтрации по индексу, видно, насколько эффективна предложенная оптимизация: производительность возрастает в пять раз. Достигается этот результат за счет того, что считывается всего 40 полных строк вместо 30 000, как в первом примере.
- Пример 3 демонстрирует ситуацию, когда подзапрос оказывается неэффективным. Оставшийся после фильтрации по индексу набор результатов так мал, что подзапрос становится более затратным, чем считывание всех данных из таблицы.

В большинстве подсистем хранения индекс может покрывать только те запросы, которые обращаются к столбцам, являющимся частью индекса. Однако InnoDB позволяет немного развить эту оптимизацию. Вспомните, что вторичные индексы InnoDB хранят в листьях значения первичного ключа. Это означает, что вторичные индексы имеют дополнительные столбцы, которые InnoDB может использовать для покрытия запросов.

Например, в таблице `sakila.actor`, использующей InnoDB, построен индекс по столбцу `last_name`, который может покрывать запросы, извлекающие столбец первичного ключа `actor_id`, хотя этот столбец технически не относится к индексу:

```
mysql> EXPLAIN SELECT actor_id, last_name
    -> FROM sakila.actor WHERE last_name = 'HOPPER'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: ref
possible_keys: idx_actor_last_name
          key: idx_actor_last_name
      key_len: 137
          ref: const
         rows: 2
        Extra: Using where; Using index
```

Улучшения в будущих версиях MySQL

Многие из упомянутых нами особенностей являются результатом ограничений API подсистемы хранения, которые не позволяют MySQL накладывать фильтры через эти API на подсистему хранения. Если бы MySQL была способна это делать, она могла бы отправить запрос к данным, вместо того чтобы вытаскивать их на сервер, где происходит анализ запроса. На момент написания книги в еще не выпущенной MySQL 5.6 сделано существенное улучшение в API подсистемы хранения, называемое *index condition pushdown* (условный индекс с магазинной памятью). Оно значительно изменит выполнение запросов и позволит отказаться от многих приемов, которые мы обсудили.

Использование просмотра индекса для сортировки

В MySQL есть два способа получения упорядоченных результатов: сортировка или просмотр индекса по порядку[1]. О том, что MySQL собирается просматривать индекс, можно узнать, обнаружив слово `index` в столбце `type` таблицы, выводимом командой `EXPLAIN` (не путайте с сообщением `Using index` в столбце `Extra`).

Просмотр самого индекса выполняется быстро, поскольку это просто требует перемещения от одной записи к другой. Однако если MySQL не использует индекс для покрытия запроса, ей приходится считывать каждую строку, которую она находит в индексе. В основном это операции ввода/вывода с произвольным доступом, поэтому чтение данных в порядке, соответствующем индексу, обычно намного медленнее, чем последовательный просмотр таблицы, особенно если рабочая нагрузка характеризуется большим объемом ввода/вывода.

MySQL может использовать один и тот же индекс как для сортировки, так и для поиска строк. По возможности стоит проектировать индексы так, чтобы они были полезны для решения обеих задач.

Сортировка результатов по индексу работает только в тех случаях, когда порядок элементов в точности соответствует порядку, указанному в разделе `ORDER BY`, а все столбцы отсортированы в одном направлении (по возрастанию или по убыванию)[2]. Если в запросе соединяются несколько таблиц, то необходимо, чтобы в разделе `ORDER BY` упоминались только столбцы из первой таблицы. Раздел `ORDER BY` имеет то же ограничение, что и поисковые запросы: должен быть указан самый левый префикс индекса. Во всех остальных случаях MySQL использует сортировку.

Есть один случай, когда в разделе `ORDER BY` не обязательно должен быть указан левый префикс индекса: если для начальных столбцов индекса в параметрах `WHERE` или `JOIN` заданы константы, они могут заполнить пропуски в индексе.

---

[1] В MySQL есть два алгоритма сортировки, они будут детально рассмотрены в главе 7.

[2] Если вы нуждаетесь в сортировке в разных направлениях, воспользуйтесь приемом, предусматривающим хранение обратных или отрицательных значений.

Например, в таблице rental в стандартной тестовой базе данных Sakila есть индекс по столбцам (rental_date, inventory_id, customer_id):

```
CREATE TABLE rental (
    ...
    PRIMARY KEY (rental_id),
    UNIQUE KEY rental_date (rental_date,inventory_id,customer_id),
    KEY idx_fk_inventory_id (inventory_id),
    KEY idx_fk_customer_id (customer_id),
    KEY idx_fk_staff_id (staff_id),
    ...
);
```

MySQL использует индекс rental_date для сортировки результатов в следующем запросе, что доказывает отсутствие упоминания о файловой сортировке[1] в результатах команды EXPLAIN:

```
mysql> EXPLAIN SELECT rental_id, staff_id FROM sakila.rental
    -> WHERE rental_date = '2005-05-25'
    -> ORDER BY inventory_id, customer_id\G
*************************** 1. row ***************************
           type: ref
possible_keys: rental_date
            key: rental_date
           rows: 1
          Extra: Using where
```

Это работает даже несмотря на то, что в условии ORDER BY указан не левый префикс индекса, поскольку мы определили условие равенства для первого столбца в индексе.

Приведем еще несколько запросов, в которых для сортировки можно использовать индекс. В следующем примере это возможно, потому что для первого столбца индекса в запросе задана константа, а в условии ORDER BY предусмотрена сортировка по второму столбцу. В совокупности получаем левый префикс индекса:

```
... WHERE rental_date = '2005-05-25' ORDER BY inventory_id DESC;
```

Следующий запрос также работает, поскольку в разделе ORDER BY указаны два столбца, образующие левый префикс индекса:

```
... WHERE rental_date > '2005-05-25' ORDER BY rental_date, inventory_id;
```

Приведем несколько примеров, в которых индекс *не может* применяться для сортировки.

- В запросе использованы два разных направления сортировки, но все столбцы индекса отсортированы по возрастанию:

  ```
  ... WHERE rental_date = '2005-05-25' ORDER BY inventory_id DESC,  customer_id ASC;
  ```

- Здесь в условии ORDER BY указан столбец, не входящий в индекс:

  ```
  ... WHERE rental_date = '2005-05-25' ORDER BY inventory_id, staff_id;
  ```

[1] MySQL называет это файловой сортировкой, но она не обязательно использует файлы.

- Здесь столбцы, заданные во фразах `WHERE` и `ORDER BY`, не образуют левый префикс индекса:

```
... WHERE rental_date = '2005-05-25' ORDER BY customer_id;
```

- Этот запрос в первом столбце содержит условие поиска по диапазону, поэтому MySQL не использует оставшуюся часть индекса:

```
... WHERE rental_date > '2CC5-C5-25' ORDER BY inventory_id, customer_id;
```

- Здесь для столбца `inventory_id` есть несколько сравнений на равенство. С точки зрения сортировки это, в сущности, то же самое, что и условие поиска по диапазону:

```
... WHERE rental_date = '2CC5-C5-25' AND inventory_id IN(1,2) ORDER BY  customer_id;
```

- В этом примере MySQL теоретически могла бы использовать индекс для сортировки соединения, но не делает этого, поскольку оптимизатор помещает таблицу `film_actor` в соединение второй по счету (способы изменения порядка соединения показаны в следующей главе):

```
mysql> EXPLAIN SELECT actor_id, title FROM sakila.film_actor
    -> INNER JOIN sakila.film USING(film_id) ORDER BY actor_id\G
+------------+---------------------------------------------+
| table      | Extra                                       |
+------------+---------------------------------------------+
| film       | Using index; Using temporary; Using filesort |
| film_actor | Using index                                 |
+------------+---------------------------------------------+
```

Одним из наиболее важных способов применения сортировки по индексу является запрос, в котором есть как раздел `ORDER BY`, так и раздел `LIMIT`. Далее мы подробнее остановимся на этом вопросе.

Упакованные (сжатые по префиксу) индексы

MyISAM использует префиксное сжатие для уменьшения размера индекса, обеспечивая таким образом размещение большей части индекса в памяти и в некоторых случаях значительное увеличение производительности. По умолчанию эта подсистема хранения упаковывает только строковые значения, но можно задать и сжатие целочисленных значений.

Для сжатия блока индекса MyISAM сохраняет первое значение полностью, а при сохранении каждого последующего значения в том же блоке записывает только количество байтов, совпадающих с частью префикса, плюс отличающиеся данные суффикса. Например, если первым значением является слово `perform`, а вторым — `performance`, то второе значение будет храниться как `7,ance`. MyISAM также может выполнять префиксное сжатие смежных указателей на строки.

Сжатые блоки занимают меньше места, но замедляют некоторые операции. Поскольку каждое значение сжатого префикса зависит от предшествующего ему значения, MyISAM не может выполнить двоичный поиск для нахождения нужного элемента и вынужден просматривать блок с самого начала. Последовательный просмотр

в прямом направлении выполняется быстро, но просмотр в обратном направлении — например, в случае ORDER BY DESC — работает хуже. Любая операция, требующая нахождения строки в середине блока, в среднем приводит к просмотру половины данных.

Эталонное тестирование показало, что при большой загрузке процессора упакованные ключи в несколько раз замедляют поиск по индексу в таблицах MyISAM из-за сканирования, необходимого для произвольного поиска. Поиск упакованного ключа в обратном направлении происходит еще медленнее. Приходится искать компромисс между потреблением процессора и памяти, с одной стороны, и дисковых ресурсов — с другой. Упакованные индексы занимают около 1/10 размера диска, поэтому при рабочей нагрузке с большим объемом ввода/вывода для некоторых запросов это часто приводит к ощутимому сокращению затрат.

Упаковкой индексов в таблице вы можете управлять с помощью параметра PACK_KEYS команды CREATE TABLE.

Избыточные и дублирующиеся индексы

MySQL позволяет создавать несколько индексов по одному и тому же столбцу; она не замечает такие ошибки и не защищает вас от них. MySQL должна обслуживать каждый дублирующийся индекс отдельно, а оптимизатор запросов в своей работе будет учитывать их все. Это может серьезно ухудшить производительность.

Дублирующимися являются индексы одного типа, созданные на основе одного и того же набора столбцов в одинаковом порядке. Старайтесь избегать их создания, а обнаружив — удаляйте.

Иногда можно создать дублирующиеся индексы, даже не подозревая об этом. Например, посмотрите на следующий код:

```
CREATE TABLE test (
    ID INT NOT NULL PRIMARY KEY,
    A INT NOT NULL,
    B INT NOT NULL,
    UNIQUE(ID),
    INDEX(ID)
) ENGINE=InnoDB;
```

Неопытный пользователь может подумать, что здесь столбец является первичным ключом, ему добавлено ограничение UNIQUE и, кроме того, создан индекс для применения в запросах. На самом же деле MySQL реализует ограничения UNIQUE и PRIMARY KEY с помощью индексов, так что здесь созданы три индекса по одному и тому же столбцу! Обычно в таких действиях нет смысла, за исключением ситуаций, когда вы хотите иметь различные типы индексов по одному столбцу для выполнения различных типов запросов[1].

Избыточные индексы несколько отличаются от дублирующихся. Если существует индекс по паре столбцов (A, B), то отдельный индекс по столбцу A будет избыточ-

[1] Индекс не обязательно является дублирующимся, если это индекс другого типа. Часто есть веские причины для одновременного наличия индексов KEY(col) и FULLTEXT KEY(col).

ным, поскольку он является префиксом первого. То есть индекс по (A, B) может быть использован и как индекс по одному столбцу A. (Такой тип избыточности относится только к индексам, упорядоченным на основе B-дерева.) Однако ни индекс по столбцам (B, A), ни индекс по столбцу B не будут избыточными, поскольку B не является левым префиксом для (A, B). Более того, индексы различных типов (например, хеш-индексы или полнотекстовые индексы) не являются избыточными по отношению к индексам, упорядоченным на основе B-дерева, независимо от того, какие столбцы они покрывают.

Избыточность обычно появляется при добавлении индексов к таблице. Например, возможно, кто-то добавит индекс по (A, B) вместо расширения существующего индекса по столбцу A для покрытия (A, B). Это может случиться и при изменении индекса таким образом, чтобы он покрывал (A, ID). Столбец ID является первичным ключом, поэтому, если вы используете InnoDB, он уже включен в индекс.

В большинстве случаев избыточные индексы не нужны, и для того, чтобы избежать их появления, стоит расширять существующие индексы, а не добавлять новые. Но все же бывают случаи, когда избыточные индексы необходимы в связи с производительностью. Расширение существующего индекса может значительно увеличить его размер и ухудшить производительность некоторых запросов.

Например, если у вас есть индекс по целочисленному столбцу и вы расширяете его длинным столбцом типа VARCHAR, он может существенно замедлиться. В частности, это относится к случаям, когда ваши запросы используют покрывающий индекс либо это таблица MyISAM и вы выполняете много просмотров по диапазону (поскольку MyISAM использует сжатие префиксов).

Вспомните таблицу userinfo, описанную в разделе «Вставка строк в порядке первичного ключа в InnoDB». Она содержит 1 миллион строк, и для каждого значения столбца state_id существует около 20 000 записей. В этой таблице есть индекс по столбцу state_id, который полезен при следующем запросе (назовем его Q1):

```
mysql> SELECT count(*) FROM userinfo WHERE state_id=5;
```

Простой эталонный тест показывает, что скорость его выполнения — почти 115 запросов в секунду (QPS). Теперь рассмотрим запрос Q2, который извлекает несколько столбцов, а не просто подсчитывает строки:

```
mysql> SELECT state_id, city, address FROM userinfo WHERE state_id=5;
```

Для этого запроса результат меньше 10 QPS[1]. Простое решение по увеличению производительности состоит в расширении индекса до трех столбцов (state_id, city, address) так, чтобы индекс покрывал запрос:

```
mysql> ALTER TABLE userinfo DROP KEY state_id,
    -> ADD KEY state_id_2 (state_id, city, address);
```

[1] В этом случае все данные помещаются в память. Если размер таблицы велик и рабочая нагрузка характеризуется большим объемом операций ввода/вывода, то различие станет гораздо заметнее. В том, что запрос COUNT() в 100 или более раз быстрее покрывающего индекса, нет ничего необычного.

После расширения индекса запрос Q2 выполняется быстрее, но Q1 становится медленнее. Если мы действительно хотим ускорить оба запроса, нам нужно оставить оба индекса, даже несмотря на то, что индекс по одному столбцу является избыточным. В табл. 5.3 показаны подробные результаты для обоих запросов и стратегий индексирования с подсистемами хранения MyISAM и InnoDB. Обратите внимание, что в InnoDB производительность запроса Q1 не так сильно падает только с индексом state_id_2, поскольку в ней не используется сжатие ключа.

Таблица 5.3. Результаты эталонных тестов для запросов SELECT с разными стратегиями индексирования, QPS

Подсистема хранения, запрос	Только state_id	Только state_id_2	И state_id, и state_id_2
MyISAM, Q1	114,96	25,40	112,19
MyISAM, Q2	9,97	16,34	16,37
InnoDB, Q1	108,55	100,33	107,97
InnoDB, Q2	12,12	28,04	28,06

Недостаток наличия двух индексов — затраты на их поддержку. В табл. 5.4 показано, сколько времени занимает вставка 1 миллиона строк в таблицу.

Таблица 5.4. Скорость вставки 1 миллиона строк при различных стратегиях индексирования, в секундах

Подсистема хранения	Только state_id	И state_id, и state_id_2
InnoDB, достаточно памяти для обоих индексов	80	136
MyISAM, достаточно памяти только для одного индекса	72	470

Как видите, вставка новых строк в таблицу с большим количеством индексов происходит медленнее. Это общее правило: добавление новых индексов может серьезно повлиять на производительность операций INSERT, UPDATE и DELETE, особенно если новый индекс не помещается в памяти.

Избыточные и дублирующиеся индексы нужно просто удалять, однако сначала следует их идентифицировать. Вы можете писать сложные запросы к таблице INFORMATION_SCHEMA, но есть два более простых способа. Можно использовать представления в пакете Шломи Ноача (Shlomi Noach) common_schema, предлагающем набор утилит и представлений, которые вы можете установить на свой сервер (http://code.google.com/p/common-schema/). Это быстрее и проще, чем писать запросы самостоятельно. Или можете использовать инструмент проверки pt-duplicate-key-checker, включенный в пакет Percona Toolkit, который анализирует структуры таблиц и выявляет дублирующиеся или избыточные индексы. Внешний инструмент — вероятно, лучший выбор для очень больших серверов: запросы к таблицам INFORMATION_SCHEMA могут вызывать проблемы с производительностью при большом количестве данных или таблиц.

Будьте осторожны, отыскивая среди индексов кандидатов на удаление или расширение. Помните, что в InnoDB индекс по столбцу `A` в нашем примере эквивалентен индексу по столбцам `(A, ID)`, потому что первичный ключ добавляется к листу вторичного индекса. Если вы выполняете запрос типа `WHERE A = 5 ORDER BY ID`, индекс будет очень полезен. Но если расширите индекс до двух столбцов `(A, B)`, тогда он на самом деле станет `(A, B, ID)` и запрос начнет использовать файловую сортировку для раздела `ORDER BY`. Целесообразно проверять планируемые изменения с помощью такого инструмента, как pt-upgrade из пакета Percona Toolkit.

Неиспользуемые индексы

Помимо дублирующихся и избыточных индексов, у вас могут быть индексы, которые сервер просто не задействует. Они лежат мертвым грузом, и вам стоит подумать о том, как избавиться от них[1]. Существует два инструмента, которые могут помочь идентифицировать неиспользуемые индексы. Возможно, самым простым и точным является таблица `INFORMATION_SCHEMA.INDEX_STATISTICS` в Percona Server и MariaDB. Просто включите переменную сервера `userstats` (по умолчанию она отключена) и на некоторое время запустите сервер. Так вы сможете узнать, насколько активно используется каждый индекс.

Кроме того, можно воспользоваться инструментом pt-index-usage, включенным в пакет Percona Toolkit. Он считывает журнал запросов, выполняет `EXPLAIN` с каждым из них и по завершении работы выводит отчет об индексах и запросах. Вы можете применять этот инструмент не только для поиска неиспользуемых индексов, но и для ознакомления с планами выполнения запросов, например, поиска похожих запросов, которые сервер иногда выполняет по-разному. Это поможет вам идентифицировать запросы, которые временами не обеспечивают должного качества сервиса, в результате чего вы сможете оптимизировать их. Кроме того, поскольку этот инструмент может хранить полученные результаты в таблицах в MySQL, вы можете обращаться к ним с помощью SQL-запросов.

Индексы и блокировки

Индексы позволяют блокировать меньше строк при выполнении запроса. Если запросы не обращаются к не нужным им строкам, то блокируется меньше строк, тем самым по двум причинам увеличивается производительность. Во-первых, даже несмотря на то, что блокировки строк в InnoDB реализованы весьма эффективно и потребляют совсем немного памяти, с ними все же сопряжены некоторые издержки. Во-вторых, блокировка большего количества строк, чем необходимо, увеличивает борьбу за блокировки и уменьшает уровень параллелизма.

[1] Некоторые индексы функционируют как ограничивающие уникальность, поэтому, даже если индекс не используется для запросов, он может применяться для предотвращения дублирования значений.

InnoDB блокирует строки только в момент доступа к ним, а индекс позволяет уменьшить количество строк, к которым обращается InnoDB, и, как следствие, тоже блокирует их. Однако это работает лишь в том случае, если InnoDB может отфильтровывать ненужные строки *на уровне подсистемы хранения*. Если индекс не позволяет InnoDB сделать это, то сервер MySQL вынужден применять условие `WHERE` после того, как InnoDB извлечет строки и вернет их серверу. К этому моменту избежать блокировки строк невозможно: InnoDB уже заблокировала их, и они останутся в таком состоянии в течение некоторого времени. В MySQL 5.1 и более новых версиях InnoDB может разблокировать строки после того, как сервер отфильтрует их, в более старых версиях MySQL InnoDB не разблокирует строки до момента завершения транзакции.

Это проще объяснить на примере. Мы снова используем демонстрационную базу данных Sakila:

```
mysql> SET AUTOCOMMIT=0;
mysql> BEGIN;
mysql> SELECT actor_id FROM sakila.actor WHERE actor_id < 5
    ->   AND actor_id <> 1 FOR UPDATE;
+----------+
| actor_id |
+----------+
|        2 |
|        3 |
|        4 |
+----------+
```

Этот запрос возвращает только строки со второй по четвертую, но на самом деле захватывает монопольные блокировки на строки с первой по четвертую. InnoDB блокирует первую строку потому, что план, который оптимизатор выбрал для этого запроса, предусматривает поиск по диапазону индекса:

```
mysql> EXPLAIN SELECT actor_id FROM sakila.actor
    -> WHERE actor_id < 5 AND actor_id <> 1 FOR UPDATE;
+----+-------------+-------+-------+---------+--------------------------+
| id | select_type | table | type  | key     | Extra                    |
+----+-------------+-------+-------+---------+--------------------------+
|  1 | SIMPLE      | actor | range | PRIMARY | Using where; Using index |
+----+-------------+-------+-------+---------+--------------------------+
```

Другими словами, низкоуровневая операция подсистемы хранения была такой: «начать с первой строки индекса и извлекать все строки до тех пор, пока выражение `actor_id < 5` не станет ложным». Сервер не сообщил подсистеме InnoDB об условии `WHERE`, которое отбрасывает первую строку. Обратите внимание на наличие сообщения `Using where` в столбце `Extra` вывода команды `EXPLAIN`. Это указывает на то, что сервер MySQL применяет фильтр `WHERE` после того, как подсистема хранения вернула строки.

Приведем еще один запрос, который доказывает, что строка 1 заблокирована, хотя она и не отображалась в результатах первого запроса. Оставьте первое подключение открытым, запустите второе соединение и выполните следующее:

```
mysql> SET AUTOCOMMIT=0;
mysql> BEGIN;
mysql> SELECT actor_id FROM sakila.actor WHERE actor_id = 1 FOR UPDATE;
```

Запрос зависнет, ожидая, пока первая транзакция не освободит блокировку в строке 1. Это поведение необходимо для корректной работы покомандной репликации (обсуждается в главе 10)[1].

Как показывает данный пример, InnoDB может блокировать строки, которые ей на деле не нужны, даже при использовании индекса. Все становится еще хуже, когда InnoDB не может использовать индекс для поиска и блокировки строк: если для запроса нет индекса, MySQL приходится выполнять полное сканирование таблицы и блокировать каждую строку независимо от того, нужна она или нет.

Есть малоизвестная подробность об InnoDB, индексах и блокировке: InnoDB может захватывать разделяемые блокировки на вторичные индексы (на чтение), но для доступа к первичному ключу необходимы монопольные блокировки (на запись). Это исключает возможность использования покрывающего индекса и может сделать команду `SELECT FOR UPDATE` значительно более медленной, чем `LOCK IN SHARE MODE` или неблокирующий запрос.

Кейсы по индексированию

Проще всего разобраться в концепциях индексирования на практических примерах, поэтому мы приготовили несколько кейсов.

Предположим, нужно спроектировать интерактивный сайт знакомств с профилями пользователей, которые состоят из различных столбцов, таких как «Страна», «Регион», «Город», «Пол», «Возраст», «Цвет глаз» и т. п. Сайт должен поддерживать поиск профиля по различным комбинациям этих свойств. Также он должен позволять пользователю выдавать нужное количество результатов о времени последнего посещения сайта владельцем профиля, его оценке другими пользователями и т. д. и сортировать их. Каким образом спроектировать индексы для таких сложных требований?

Как ни странно, в первую очередь нужно решить, будем ли мы использовать сортировку с помощью индексов или подойдет обычная сортировка. Сортировка с помощью индексов налагает ограничения на построение индексов и запросов. Например, мы не можем использовать индекс для условия `WHERE age BETWEEN 18 AND 25`, если в том же самом запросе индекс применяется для сортировки пользователей по оценкам, полученным от других пользователей. Если MySQL задействует в запросе индекс для поиска по диапазону, то она не может использовать другой индекс (или суффикс того же самого индекса) с целью упорядочения. Учитывая, что это будет одним из наиболее распространенных условий раздела `WHERE`, мы считаем само собой разумеющимся, что для многих запросов потребуется обычная (то есть файловая) сортировка.

---

[1] Хотя сервер не может блокировать строки на некоторых уровнях изоляции транзакций при использовании двоичного журналирования на основе строк, на практике сложно добиться желаемого поведения. Даже в MySQL 5.6.3 с изоляцией коммита операций чтения и журналированием на основе строк приведенный пример вызовет блокировку.

Поддержка нескольких видов фильтрации

Теперь нужно посмотреть, какие столбцы содержат много различных значений и какие столбцы появляются в разделе `WHERE` чаще всего. Индексы по столбцам с большим количеством различных значений будут высокоселективны. Обычно это хорошо, поскольку высокая селективность позволяет MySQL эффективнее отфильтровывать ненужные строки.

Столбец `country`, видимо, не будет селективным, но, скорее всего, будет появляться в большинстве запросов. Селективность столбца `sex`, несомненно, низка, но он, вероятно, будет присутствовать в каждом запросе. Имея это в виду, создадим набор индексов для нескольких сочетаний столбцов с префиксом по двум столбцам (`sex, country`).

Считается, что бесполезно индексировать столбцы с очень низкой селективностью. Так почему же мы поместили неселективный столбец в начале каждого индекса? В своем ли мы уме?

Для этого у нас есть две причины. Первая заключается в том, что, как уже указывалось, почти в каждом запросе будет использоваться столбец `sex`. При проектировании сайта мы можем даже сделать так, что пользователи будут выполнять поиск, только включив пол в состав критериев. Но что важнее, добавление этого столбца не принесет особых потерь, поскольку у нас есть туз в рукаве.

Вот в чем заключается трюк: даже если запрос не выполняет фильтрацию по столбцу `sex`, мы можем обеспечить использование этого индекса, добавив в раздел `WHERE` выражение `AND sex IN('m', 'f')`. На деле это выражение не будет фильтровать строки, поэтому функциональность запроса будет точно такой же, как и при отсутствии столбца `sex` во фразе `WHERE`. Однако нам нужно включить этот столбец, поскольку это позволит MySQL использовать больший префикс индекса. Данный прием полезен в ситуациях, подобных описанной ранее, но он будет плохо работать, если столбец содержит много различных значений, поскольку список `IN()` станет слишком большим.

Этот случай иллюстрирует общий принцип: учитывайте все варианты. Разрабатывая индексы, думайте не только об их оптимизации под существующие запросы, но и об оптимизации запросов под эти индексы. Если вы видите необходимость в индексе, но считаете, что он может негативно повлиять на некоторые запросы, подумайте, нельзя ли изменить запросы. Оптимизируйте запросы и индексы одновременно, стараясь найти компромисс, — нельзя спроектировать хорошую схему индексирования в вакууме.

Далее нужно подумать о том, какие еще могут встретиться комбинации условий `WHERE`, и решить, какие из них без правильного индексирования окажутся медленными. Индекс по столбцам (`sex, country, age`) является очевидным выбором, также, скорее всего, потребуются индексы по столбцам (`sex, country, region, age`) и (`sex, country, region, city, age`).

Таким образом, у нас появляется очень много индексов. Если мы хотим использовать одни и те же индексы в запросах разного типа и не создавать слишком много ком-

бинаций разных условий, то можем задействовать вышеупомянутый прием с `IN()` и избавиться от индексов по столбцам `(sex, country, age)` и `(sex, country, region, age)`. Если они не указаны в форме поиска, можно сделать так, чтобы на префиксные столбцы индекса налагались ограничения равенства, и определить список всех стран или всех регионов страны (объединенные списки всех стран, всех регионов и всех полов, вероятно, окажутся слишком большими).

Эти индексы будут полезны для наиболее часто задаваемых критериев поиска, но как разработать индексы для реже встречающихся вариантов, таких как `has_pictures`, `eye_color`, `hair_color` и `education`? Если эти столбцы не являются высокоселективными и используются нечасто, мы можем просто пропустить их и позволить MySQL сканировать некоторое количество дополнительных строк. В качестве альтернативы можно добавить их перед столбцом `age` и использовать прием с `IN()`, чтобы обрабатывать случаи, когда они не указаны.

Возможно, вы заметили, что мы помещаем столбец `age` в конец индекса. Что особенного в этом столбце и почему он должен располагаться именно там? Мы хотим, чтобы MySQL использовала как можно больше столбцов индекса, поскольку она задействует только левый префикс до первого условия, задающего диапазон значений, включительно. Для всех остальных столбцов из раздела `WHERE` задается сравнение на равенство, но для `age` почти наверняка понадобится поиск по диапазону (например, `age BETWEEN 18 AND 25`).

Мы могли бы преобразовать это условие в список `IN()`, например `age IN(18, 19, 20, 21, 22, 23, 24, 25)`, но это не всегда возможно для запросов такого типа. Общий принцип, который мы пытаемся проиллюстрировать, состоит в том, что столбец, для которого может быть задано условие поиска по диапазону, следует помещать в конец индекса, тогда оптимизатор будет максимально использовать индекс.

Мы уже говорили, что можно добавлять в индекс все больше и больше столбцов и использовать списки `IN()` в тех случаях, когда эти столбцы не являются частью запроса `WHERE`, но есть вероятность перестараться и столкнуться с проблемами. Применение слишком большого количества списков вызывает лавинообразный рост числа комбинаций, которые оптимизатор должен оценить, и может значительно снизить скорость выполнения запроса. Взгляните на следующий пример запроса `WHERE`:

```
WHERE eye_color   IN('brown','blue','hazel')
   AND hair_color IN('black','red','blonde','brown')
   AND sex        IN('M','F')
```

Оптимизатор преобразует это в $4 \cdot 3 \cdot 2 = 24$ комбинации, и запросу `WHERE` придется проверить каждую из них. Двадцать четыре комбинации — это не слишком много, но будьте осторожны, если их количество приблизится к 1000. В старых версиях MySQL возникали серьезные сложности с большим числом комбинаций `IN()`: оптимизация запроса занимала больше времени, чем его выполнение, и требовала значительного объема памяти. Последние версии MySQL прекращают вычисление комбинаций, если их количество становится слишком велико, и это обстоятельство ограничивает возможность использования индекса в MySQL.

Устранение дополнительных условий поиска по диапазону

Предположим, имеется столбец `last_online` и мы хотим показывать тех пользователей, которые заходили на сайт в течение предыдущей недели:

```
WHERE eye_color    IN('brown','blue','hazel')
   AND hair_color  IN('black','red','blonde','brown')
   AND sex         IN('M','F')
   AND last_online > DATE_SUB(NOW(), INTERVAL 7 DAY)
   AND age         BETWEEN 18 AND 25
```

Что такое условие поиска по диапазону?

Из вывода команды `EXPLAIN` иногда трудно понять, просматривает ли MySQL диапазон значений или список значений — `EXPLAIN` использует один и тот же термин `range` в обоих случаях. Например, MySQL указывает для следующего запроса тип `range`, как видно из строки `type`:

```
mysql> EXPLAIN SELECT actor_id FROM sakila.actor
    -> WHERE actor_id > 45\G
*************************** 1. row ***************************
           id:    1
  select_type:    SIMPLE
        table:    actor
         type:    range
```

А что скажете об этом?

```
mysql> EXPLAIN SELECT actor_id FROM sakila.actor
    -> WHERE actor_id IN(1, 4, 99)\G
*************************** 1. row ***************************
           id:    1
  select_type:    SIMPLE
        table:    actor
         type:    range
```

Глядя на вывод команды `EXPLAIN`, невозможно увидеть разницу, но мы различаем поиск по диапазону и множественные условия на равенство. В нашей терминологии второй запрос представляет собой условие множественного равенства.

Это не придирка: упомянутые два типа доступа к индексам выполняются по-разному. Условие диапазона заставляет MySQL игнорировать все дальнейшие столбцы индекса, а условие множественного равенства не налагает таких ограничений.

С этим запросом у нас будет проблема: в нем есть два условия на вхождение в диапазон. MySQL может использовать либо критерий по столбцу `last_online`, либо критерий по столбцу `age`, но не оба сразу.

Если ограничение по столбцу `last_online` появится без ограничения по столбцу `age` или столбец `last_online` более селективен, чем столбец `age`, то мы можем добавить

в конец другой набор индексов со столбцом `last_online`. Но что, если мы не можем преобразовать столбец `age` в список `IN()`, а нам и в самом деле необходимо увеличить скорость при одновременной фильтрации по столбцам `last_online` и `age`? На текущий момент не существует прямого способа сделать это, однако можно преобразовать один из диапазонов в сравнение на равенство. Для этого добавим столбец `active`, значения которого будут вычисляться периодически запускаемым заданием. Когда пользователь заходит на сайт, мы записываем в столбец значение `1`, а если он отсутствует уже в течение семи дней, то периодическое задание будет заносить в столбец значение `0`.

Подобный подход позволяет MySQL использовать такие индексы, как `(active, sex, country, age)`. Столбец может оказаться не совсем точным, но запросу такого типа высокая точность и не нужна. Если все-таки требуется точность, мы можем оставить условие `last_online` во фразе `WHERE`, но *не индексировать этот столбец*. Данный прием подобен тому, который мы использовали для имитации хеш-индексов в процессе поиска адресов URL ранее в этой главе. Для фильтрации по этому условию индекс не используется, но, поскольку маловероятно, что будет отброшено много найденных по данному индексу строк, его наличие все равно не дало бы ощутимого эффекта. Иными словами, от отсутствия этого индекса производительность запроса заметно не пострадает.

Теперь вы, вероятно, поняли суть: если нужно увидеть и активных, и неактивных пользователей, можно добавить список `IN()`.

Мы добавляли много таких списков, но в качестве альтернативы можно создавать отдельные индексы, удовлетворяющие любым комбинациям столбцов, по которым хотим осуществлять фильтрацию. Мы бы использовали по крайней мере такие индексы: `(active, sex, country, age)`, `(active, country, age)`, `(sex, country, age)` и `(country, age)`.

Хотя перечисленные индексы порой оказываются более эффективными для каждого конкретного запроса, затраты на обслуживание их всех в сочетании с требующимся им дополнительным пространством делают подобную стратегию сомнительной.

Это тот случай, когда модификация оптимизатора может реально повлиять на выбор эффективной стратегии индексирования. Если в следующей версии MySQL научится выполнять непоследовательный просмотр индекса, можно будет использовать несколько условий поиска по диапазону в одном индексе. Тогда нам не потребуются списки `IN()` для рассмотренных здесь типов запросов.

Оптимизация сортировки

Последний вопрос, которого мы хотим коснуться в этом кейсе, — сортировка.

Файловая сортировка небольшого результирующего набора происходит быстро, но что, если запрос находит миллионы строк? Например, что будет, если в разделе `WHERE` будет присутствовать только столбец `sex`?

Мы можем добавить специальные индексы для сортировки таких случаев с низкой селективностью. Например, индекс по столбцам (`sex, rating`) можно использовать для следующего запроса:

```
mysql> SELECT <столбцы> FROM profiles WHERE sex='M' ORDER BY rating LIMIT 10;
```

Этот запрос содержит и раздел `ORDER BY`, и раздел `LIMIT`, но без индекса он будет очень медленно работать.

Даже с индексом запрос может оказаться медленным, если интерфейс пользователя предусматривает разбиение на страницы и кто-то запрашивает страницу, находящуюся далеко от начала. Вот пример неудачного сочетания разделов `ORDER BY` и `LIMIT` со смещением:

```
mysql> SELECT <столбцы> FROM profiles WHERE sex='M' ORDER BY rating LIMIT 100000, 10;
```

Такие запросы могут оказаться серьезной проблемой независимо от того, как они проиндексированы, поскольку из-за большого смещения они должны тратить основное время на просмотр значительного количества строк, которые потом будут отброшены. Денормализация, предварительное вычисление и кэширование являются, по всей видимости, единственными стратегиями, подходящими для обработки подобных запросов. Еще лучшей стратегией можно назвать ограничение количества страниц, которые вы позволяете просматривать пользователю. Маловероятно, что это повлияет на его возможности, поскольку никто не станет просматривать десятитысячную страницу результатов поиска.

Еще одной хорошей стратегией оптимизации таких запросов является использование отложенного соединения. Под этим термином мы понимаем использование покрывающего индекса для извлечения только столбцов первичного ключа тех строк, которые нужны в итоге. Затем вы можете снова соединить их с таблицей для извлечения всех нужных столбцов. Это помогает минимизировать объем работы, выполняемой MySQL в процессе сбора данных, которые она затем отбросит. Приведем пример, требующий для большей эффективности наличия индекса по столбцам (`sex, rating`):

```
mysql> SELECT <столбцы> FROM profiles INNER JOIN (
    ->    SELECT <первичный ключ> FROM profiles
    ->    WHERE x.sex='M' ORDER BY rating LIMIT 100000, 10
    -> ) AS x USING(<первичный ключ>);
```

Обслуживание индексов и таблиц

Вы создали таблицы с нужными типами данных и добавили индексы, однако работа еще не закончена: таблицы и индексы требуется обслуживать для обеспечения нормальной работы. Главными задачами обслуживания таблицы являются поиск и устранение повреждений, поддержка точной статистики по индексам и уменьшение фрагментации.

Поиск и исправление повреждений таблицы

Самое плохое, что может произойти с таблицей, — она может быть повреждена. С таблицами MyISAM такое часто происходит при аварийном останове. Однако во всех подсистемах хранения могут возникать повреждения индексов из-за проблем с оборудованием, внутренних программных ошибок MySQL или операционной системы.

Повреждение индексов может привести к тому, что запросы будут возвращать неправильные результаты, вызывать ошибки дублирования ключа, хотя дубликатов нет, и даже приводить к блокировкам и аварийным остановкам. Если вы столкнулись со странным поведением, например ошибками, которых, по вашему мнению, просто не может быть, — запустите команду `CHECK TABLE`, чтобы выяснить, не повреждена ли таблица (обратите внимание на то, что одни подсистемы хранения не поддерживают эту команду, а другие поддерживают многочисленные параметры, позволяющие указать, насколько тщательно нужно проверить таблицу). Команда `CHECK TABLE` обычно выявляет большинство ошибок в таблицах и индексах.

Исправить поврежденную таблицу позволяет команда `REPAIR TABLE`, но и ее поддерживают не все подсистемы хранения. В таком случае можно выполнить пустую команду `ALTER`, например, просто указав подсистему, которая и так уже используется для таблицы. Приведем пример для таблицы типа InnoDB:

```
mysql> ALTER TABLE innodb_tbl ENGINE=INNODB;
```

В качестве альтернативы можно либо использовать офлайновую утилиту восстановления для этой подсистемы хранения, например myisamchk, либо выгрузить все данные в файл, а затем загрузить обратно. Но если повреждение зафиксировано в системной области или в области строк таблицы, а не в индексе, вы, скорее всего, не сможете использовать ни один из этих вариантов. В этом случае, возможно, останется лишь восстановить таблицу из резервной копии или попытаться восстановить данные из поврежденных файлов.

Если вы столкнулись с повреждением в подсистеме хранения InnoDB, значит, произошло что-то серьезное и нужно немедленно с этим разобраться. InnoDB просто не должна повреждаться. Она спроектирована очень устойчивой к повреждениям. Повреждение свидетельствует о проблеме с оборудованием, например об ошибках памяти или дисков (вероятно), об ошибках администрирования, например при управлении файлами базы данных извне (вероятно), или программной ошибке InnoDB (маловероятно). Чаще всего это такие ошибки, как, например, попытка создания резервных копий с помощью rsync. Запроса, которого стоило бы избегать, потому что он может повредить данные InnoDB, просто не существует. Вы никак не сможете выстрелить себе в ногу. Если вы нарушаете данные InnoDB, делая к ним запросы, это программная ошибка InnoDB, а не ваша вина.

Если вы столкнулись с повреждением данных, следует попытаться определить, почему это произошло, — не стоит просто восстанавливать данные, поскольку

повреждение может повториться. Вы можете восстановить данные, запустив InnoDB в режиме принудительного восстановления с параметром `innodb_force_recovery` (см. руководство пользователя по MySQL). Или использовать пакет с открытым исходным кодом Percona InnoDB Data Recovery Toolkit (http://www.percona.com/software/mysql-innodb-data-recovery-tools/) для извлечения данных непосредственно из поврежденных файлов.

Обновление статистики индекса

Для принятия решения о том, как применять индексы, оптимизатор запросов MySQL использует два вызова API, чтобы узнать у подсистемы хранения, каким образом распределены значения в индексе. Первым является вызов `records_in_range()`, который получает границы диапазона и возвращает количество записей в нем. Для некоторых подсистем хранения, например MySQL, результат будет точным, а для InnoDB — приблизительным.

Второй вызов, `info()`, может возвращать данные различных типов, включая кардинальность индекса (приблизительное количество записей имеется для каждого значения ключа).

Если подсистема хранения не сообщает оптимизатору точную информацию о количестве строк, которые должен будет просмотреть запрос, или если план запроса слишком сложен и количество строк будет варьироваться на разных этапах его выполнения, оптимизатор использует для оценки этой величины статистику по индексам. Оптимизатор MySQL ориентирован на затраты, а основным показателем затрат является количество данных, к которым будет обращаться запрос. Если статистика не была сгенерирована или данные устарели, оптимизатор может принимать неправильные решения. Выход состоит в запуске команды `ANALYZE TABLE`.

Каждая подсистема хранения по-своему реализует статистику по индексам, поэтому частота, с которой придется запускать команду `ANALYZE TABLE`, а также затраты на ее команды, различаются.

- Подсистема Memory не хранит статистику по индексам.
- MyISAM сохраняет статистику на диске, а команда `ANALYZE TABLE` выполняет полное сканирование индекса для вычисления кардинальности. На это время вся таблица блокируется.
- InnoDB не сохраняет статистику на диске, как MySQL 5.5, а вместо этого оценивает ее с помощью случайной выборки из индекса и хранит в памяти.

Кардинальность ваших индексов можно определить с помощью команды `SHOW INDEX FROM`, например:

```
mysql> SHOW INDEX FROM sakila.actor\G
*************************** 1. row ***************************
        Table: actor
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 1
```

```
  Column_name: actor_id
    Collation: A
  Cardinality: 200
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
*************************** 2. row ***************************
        Table: actor
   Non_unique: 1
     Key_name: idx_actor_last_name
 Seq_in_index: 1
  Column_name: last_name
    Collation: A
  Cardinality: 200
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
```

Эта команда выдает довольно много информации об индексах, подробно описанной в руководстве по MySQL. Однако мы все-таки хотим обратить ваше внимание на строку `Cardinality`. Она показывает, сколько различных значений подсистемой хранения обнаружено в индексе. Эти данные вы также можете получить из таблицы `INFORMATION_ SCHEMA.STATISTICS` в MySQL 5.0 и более новых версиях, что может быть очень удобно. Например, для нахождения индексов с низкой селективностью можно написать запросы к таблицам `INFORMATION_SCHEMA`. Однако имейте в виду, что эти таблицы метаданных могут сильно нагружать сервер, если количество данных на нем велико.

Статистику InnoDB стоит изучить подробнее. Она создается путем выборки нескольких случайных страниц в индексе в предположении, что остальная часть индекса выглядит аналогично. В старых версиях InnoDB выбирались восемь страниц, но в более поздних версиях количество можно устанавливать с помощью переменной `innodb_stats_sample_pages`. Если значение превысит 8, то теоретически это поможет генерировать более репрезентативную статистику индекса, особенно для очень больших таблиц. Однако может случиться и по-другому.

InnoDB вычисляет статистику индексов при первом открытии таблиц, запуске команды `ANALYZE TABLE`, а также при значительном изменении размера таблицы (изменении размера на 1/16 или вставке 2 миллиардов строк в зависимости от того, что произойдет раньше).

InnoDB также вычисляет статистику при выполнении запросов к некоторым таблицам `INFORMATION_SCHEMA`, команд `SHOW TABLE STATUS` и `SHOW INDEX`, а еще когда клиент командной строки MySQL активизирует автозаполнение. Это может стать довольно серьезной проблемой на больших серверах со значительным количеством данных или при медленном вводе/выводе. Клиентские программы или средства мониторинга, которые делают выборку, могут провоцировать много блокировок и большую нагрузку на сервер, а также раздражать пользователей медленным временем запуска.

Вы не сможете увидеть статистику индекса, не изменив их, поскольку команда `SHOW INDEX` будет обновлять информацию. Чтобы избежать этих проблем, можно отключить параметр `innodb_stats_on_metadata`.

Если вы используете Percona Server, который включает Percona XtraDB вместо стандартного InnoDB, можете настроить его дальнейшие действия. Параметр `innodb_stats_auto_update` позволяет отключить автоматическую повторную выборку. Тем самым сбор статистики будет приостановлен, если вы не запустите команду `ANALYZE TABLE` вручную. Это может помочь, если вы боретесь с нестабильными планами запросов. Мы создали эту функцию по просьбе некоторых клиентов с очень крупными развертываниями.

Добиться значительного повышения стабильности плана запросов и более быстрого прогрева системы вы можете, используя системную таблицу для хранения статистики индекса таким образом, чтобы она не менялась при перезагрузке сервера и не нуждалась в перерасчете, уже при первом открытии InnoDB таблицы после загрузки Эта функция доступна в Percona Server 5.1 и MySQL 5.6. Функцию в Percona Server включает параметр `innodb_ use_sys_stats_table`, а в MySQL 5.6 статистику индекса контролирует параметр `innodb_analyze_is_persistent`.

Если вы отключите автоматическое обновление статистики на сервере, придется делать это вручную, периодически выполняя команду `ANALYZE TABLE` (конечно, если вы не уверены, что статистика не изменится настолько, что это ухудшит планы выполнения запросов).

Уменьшение фрагментации индекса и данных

Индексы, упорядоченные на основе B-дерева, могут становиться фрагментированными, что ухудшает производительность. Фрагментированные индексы бывают плохо заполнены и/или располагаются на диске непоследовательно.

Индексы, упорядоченные на основе B-дерева, требуют произвольного доступа к диску для «погружения» к листьям, поэтому произвольный доступ здесь является правилом, а не исключением. Однако производительность работы с листьями можно улучшить, когда они являются физически последовательными и плотно упакованными. Если это не так, то мы говорим, что они фрагментированы, и в подобном случае поиск по диапазону и полное сканирование индекса могут оказаться во много раз медленнее. Это особенно справедливо для покрывающих индексы запросов.

Фрагментации подвержены и данные, хранящиеся в таблицах. Однако механизм фрагментации данных сложнее, чем механизм фрагментации индексов. Существуют три типа фрагментации данных.

- *Фрагментация строки.* Наблюдается, когда строка хранится в виде нескольких фрагментов в разных местах. Уменьшает производительность, даже если запросу требуется только одна строка из индекса.
- *Межстрочная фрагментация.* Наблюдается, когда логически последовательные страницы или строки хранятся на диске непоследовательно. Влияет на такие опе-

рации, как полное сканирование таблицы и сканирование диапазона кластерного индекса, которые обычно выигрывают от последовательного размещения данных на диске.

- *Фрагментация свободного пространства*. Возникает, когда на страницах данных много пустого пространства. Заставляет сервер читать много данных, которые ему не нужны, впустую затрачивая время.

Таблицы MyISAM могут страдать от фрагментации всех типов, но InnoDB никогда не фрагментирует короткие строки.

Чтобы дефрагментировать данные, можно либо запустить команду `OPTIMIZE TABLE`, либо выгрузить данные в файл и заново загрузить их в базу. Эти методы работают в большинстве подсистем хранения. В некоторых из них, например в MyISAM, они также дефрагментируют индексы с помощью алгоритма сортировки, в результате чего индексы оказываются отсортированными. В старых версиях InnoDB способа дефрагментации индексов не существует, но в более новых, которые обладают способностью удалять и создавать индексы онлайн без перестраивания всей таблицы, вы можете удалить и заново создать индексы, тем самым дефрагментировав их.

В подсистемах хранения, которые не поддерживают команду `OPTIMIZE TABLE`, вы можете перестроить таблицу с помощью пустой команды `ALTER TABLE`. Достаточно указать ту же подсистему хранения, которая используется для таблицы в данный момент:

```
mysql> ALTER TABLE <таблица> ENGINE=<подсистема>;
```

В Percona Server с включенной функцией `expand_fast_index_creation` перестройка таблицы таким образом дефрагментирует таблицы и индексы InnoDB. В стандартной MySQL будет дефрагментирована только таблица (кластерный индекс). Вы можете эмулировать функциональность Percona Server, удалив все индексы, перестроив таблицу и затем вернув в нее индексы.

Не считайте, что вам обязательно нужно дефрагментировать индексы и таблицы, — сначала проверьте это. В Percona XtraBackup есть параметр `--stats`, который запускается в режиме без резервного копирования. В этом режиме распечатывается статистика индексов и таблиц, включая объем данных и свободное пространство на страницах. Это один из способов узнать, насколько фрагментированы данные. Также подумайте о том, не размещены ли данные настолько хорошо, что, упаковав их еще плотнее, вы нарушите их устойчивое состояние, в результате чего последующие обновления повлекут за собой множество случаев разделения страниц, что может плохо влиять на производительность до тех пор, пока данные вновь не придут в устойчивое состояние.

Итоги главы

Индексирование — сложная тема! Организация доступа к данным в MySQL и подсистемах хранения данных вместе со свойствами индексов делает последние очень мощным и гибким инструментом, управляющим доступом к данным как на диске, так и в памяти.

В большинстве случаев вы будете использовать в MySQL индексы, упорядоченные на основе B-дерева. Другие типы индексов лучше подходят для конкретных задач. Мы больше не будем останавливаться на них в этой главе, но стоит напомнить свойства и способы использования индексов, упорядоченных на основе B-дерева.

Приведем три принципа, которые следует учитывать при выборе индексов и написании применяющих их запросов.

1. Однострочный доступ — операция медленная, особенно в дисковом хранилище. (Твердотельные диски работают быстрее при произвольном вводе/выводе, но принцип остается справедливым.) Если сервер считывает блок данных из хранилища, а затем обращается только к одной строке, он тратит много времени впустую. Гораздо лучше считывать из блока, содержащего много нужных вам строк. Используйте индексы, чтобы локализовать ссылки, что повысит эффективность.
2. Доступ к упорядоченным строкам происходит быстро по двум причинам. Во-первых, последовательный ввод/вывод не требует поиска на диске, поэтому он быстрее, чем произвольный ввод/вывод, особенно в дисковом хранилище. Во-вторых, если сервер может считывать данные в требующемся вам порядке, ему не нужно в дальнейшем сортировать их, а у запросов `GROUP BY` не возникает необходимости сортировать и группировать строки для вычисления агрегатных значений.
3. Доступ только по индексу выполняется быстро. Если индекс содержит все столбцы, необходимые для запроса, подсистеме хранения не нужно искать другие столбцы, просматривая строки в таблице. Это позволяет избежать множества операций однострочного доступа, который, как мы знаем из пункта 1, работает медленно.

В общем, попытайтесь выбрать индексы и написать запросы так, чтобы избежать однострочного поиска, использовать встроенный порядок данных, позволяющий избавиться от операций сортировки, и применять доступ только для индексирования. Это соответствует трехзвездочной системе ранжирования, изложенной в книге Лахденмаки и Лича, упомянутой в начале главы.

Было бы здорово создавать идеальные индексы для каждого запроса к таблицам. К сожалению, иногда для этого требуется так много индексов, что это оказывается непрактичным. А в ряде случаев просто невозможно создать трехзвездочный индекс для заданного запроса (например, если запрос делается по двум столбцам, данные в одном из которых упорядочены по возрастанию, а в другом — по убыванию). В этих ситуациях вы должны соглашаться на лучший из возможных вариантов либо воплощать альтернативные стратегии, такие как денормализация или сводные таблицы.

Очень важно понимать, как именно работают индексы, и выбирать их на основе этого понимания, а не опираясь на эмпирические или эвристические правила, такие как «размещайте наиболее селективные столбцы в начале многостолбцовых индексов» или «вы должны индексировать все столбцы, которые появляются в разделе `WHERE`».

Как узнать, хорошо ли индексирована схема? Как всегда, предлагаем ответить на этот вопрос, основываясь на времени отклика. Найдите запросы, которые выполняются

слишком долго или порождают слишком большую нагрузку на сервер (подробнее об этом — в главе 3). Исследуйте схему, SQL и структуру индексов для запросов, требующих внимания. Определите, должен ли запрос проверять слишком много строк, выполнять сортировку после извлечения данных, использовать временные таблицы, получать доступ к данным с помощью произвольного ввода/вывода или искать полные строки из таблицы для извлечения столбцов, не включенных в индекс. Если вы найдете запрос, который не выигрывает от перечисленных преимуществ индексов, посмотрите, можно ли создать лучший индекс, чтобы повысить производительность. Если нет, возможно, стоит изменить запрос так, чтобы он мог использовать индекс, который либо уже существует, либо может быть создан. Об этом поговорим в следующей главе.

А если запрос не отображается в анализе времени отклика, описанном в главе 3? Не может ли случиться, что плохой запрос ускользнет от вашего внимания, даже если для его работы действительно нужен лучший индекс? Как правило, нет. Если профилирование не находит такого запроса, ничего страшного. Однако проблема может всплыть в будущем, после изменения приложения, данных и рабочей нагрузки, поэтому все же стоит найти запросы, которые не задействуют индексы, и исправить их до того, как возникнут серьезные проблемы. Используйте функции просмотра запросов в утилите pt-query-digest, это поможет вам заметить новые запросы и изучить их планы `EXPLAIN`.

6 Оптимизация производительности запросов

В предыдущей главе мы рассказали об оптимизации и индексировании схемы. Такая оптимизация является одним из необходимых условий достижения высокой производительности. Но только этого недостаточно — следует еще правильно создавать запросы. Если запрос составлен плохо, то даже самые лучшие схема и индексы не помогут.

Оптимизация запросов, индексов и схемы идут рука об руку. Постепенно набираясь опыта в написании запросов в MySQL, вы начнете понимать, как проектировать таблицы и индексы для поддержки эффективных запросов. И наоборот, знания о создании оптимальных схем будут положительно влиять на ваши запросы. Все это, конечно, придет не сразу, поэтому рекомендуем возвращаться к трем предыдущим главам по мере освоения нового материала.

Данная глава начинается с общих замечаний о конструировании запросов — на что в первую очередь обращать внимание, если запрос выполняется неэффективно. Затем мы углубимся в механизмы оптимизации запросов и детали внутреннего устройства сервера. Поможем узнать, как именно MySQL выполняет конкретный запрос и как можно изменить план его выполнения. Наконец, рассмотрим несколько случаев, когда MySQL не слишком хорошо оптимизирует запросы, и разберем типичные паттерны, помогающие MySQL более эффективно выполнять запросы.

Наша цель состоит в том, чтобы вы тщательнее разобрались в механизмах выполнения запросов, понимали, что считать эффективным, а что — нет, использовали сильные стороны MySQL и избегали слабых.

Почему запросы бывают медленными

Пытаясь писать быстрые запросы, помните, что речь идет о времени отклика. Запросы — это задачи, но они состоят из подзадач, а их выполнение требует времени. Чтобы оптимизировать запрос, вы должны оптимизировать его подзадачи. Для достижения

этой цели подзадачи можно вообще устранить, сделав так, чтобы они выполнялись реже или оказались более быстрыми[1].

Что это за подзадачи, которые нужны MySQL для выполнения запроса, и какие из них медленные? Мы не можем привести здесь полный список, но если вы выполните профилирование запроса, как было показано в главе 3, то узнаете, какие задачи он решает. В общем, можете думать о времени жизни запроса, мысленно следуя за ним от клиента к серверу, где он анализируется, планируется и выполняется, а затем вновь к клиенту. Выполнение является одним из наиболее важных этапов жизни запроса. Оно включает в себя множество обращений к подсистеме хранения для извлечения строк, а также операций, протекающих после извлечения, таких как группировка и сортировка.

Выполняя все эти задачи, запрос затрагивает сеть, процессор, выполняет такие операции, как сбор статистики, планирование и блокировка (ожидание мьютекса), и, что наиболее важно, вызывает подсистему хранения для извлечения строк. На эти вызовы затрачивается время в ходе работы с памятью, работы процессора и особенно операций ввода/вывода, если данных нет в памяти. Кроме того, в зависимости от подсистемы хранения может быть задействовано множество переключений контекста и/или системных вызовов.

Дополнительное время может быть потрачено и из-за того, что операции выполняются без необходимости, слишком много раз или чересчур медленно. Цель оптимизации состоит в том, чтобы избежать этого, устраняя операции, уменьшая их количество или делая их быстрее.

Опять же это неполная или неточная картина жизни запроса. Наша цель — показать, как важно понимать, в чем заключается жизненный цикл запроса, чтобы вы могли учитывать в работе все процессы, требующие затрат времени. Помня об этом, перейдем к методам оптимизации запросов.

Основная причина замедления: оптимизируйте доступ к данным

Главная причина, из-за которой запрос может выполняться медленно, — слишком большой объем обрабатываемых данных. Безусловно, встречаются запросы, которые по своей природе должны перерабатывать очень много всевозможных значений, и тут ничего не поделаешь. Но это довольно редкая ситуация — большинство плохих запросов можно изменить так, чтобы они обращались к меньшему

[1] Иногда вам может потребоваться изменить запрос, чтобы уменьшить его негативное влияние на другие запросы, запущенные в системе. В этом случае нужно уменьшить объем потребляемых запросом ресурсов (эта тема обсуждалась в главе 3).

объему данных. Мы полагаем, что анализировать медленно выполняющийся запрос следует в два этапа.

1. Выяснить, не извлекает ли *приложение* больше данных, чем нужно. Обычно это означает, что слишком велико количество отбираемых строк, но не исключено, что отбираются также лишние столбцы.
2. Выяснить, не анализирует ли *сервер MySQL* больше строк, чем необходимо.

Не запрашиваете ли вы лишние данные у базы?

Иногда запрос отбирает больше данных, чем необходимо, а потом отбрасывает некоторые из них. Это требует от сервера MySQL дополнительной работы, приводит к росту расходов на передачу по сети[1], а также увеличивает потребление памяти и процессорного времени на стороне сервера приложений.

Приведем несколько типичных ошибок.

- *Выбор лишних строк.* Широко распространено заблуждение, будто MySQL передает результаты по мере необходимости, а не формирует и возвращает весь результирующий набор целиком. Подобную ошибку мы часто встречали в приложениях, написанных людьми, привыкшими работать с другими СУБД. Этим разработчикам была свойственна такая методика: выполнить команду `SELECT`, которая возвращает много строк, затем выбрать *N* первых строк и закрыть результирующий набор (например, отобрать 100 последних по времени статей на новостном сайте, хотя на начальной странице нужно показать только десять). Они полагают, что MySQL вернет первые десять строк, после чего прекратит выполнение запроса. На самом деле MySQL генерирует весь результирующий набор. Затем клиентская библиотека, получив полный набор данных, большую часть отбросит. Вместо этого следовало бы включить в запрос слово `LIMIT`.
- *Выбор всех столбцов из соединения нескольких таблиц.* Если нужно отобрать всех актеров, снимавшихся в фильме *Academy Dinosaur*, не стоит писать такой запрос:

```
mysql> SELECT * FROM sakila.actor
    -> INNER JOIN sakila.film_actor USING(actor_id)
    -> INNER JOIN sakila.film USING(film_id)
    -> WHERE sakila.film.title = 'Academy Dinosaur';
```

Он возвращает все столбцы из всех трех таблиц. Лучше напишите этот запрос следующим образом:

```
mysql> SELECT sakila.actor.* FROM sakila.actor...;
```

[1] Особенно велики сетевые издержки, когда приложение и сервер работают на разных компьютерах, но даже передача данных между MySQL и приложением на одной и той же машине не обходится без этого.

- *Выбор всех столбцов.* Вы всегда должны настораживаться, встречая команду `SELECT *`. Неужели действительно нужны все столбцы без исключения? Скорее всего, нет. Выбор всех столбцов может воспрепятствовать применению таких методов оптимизации, как использование покрывающих индексов, и к тому же увеличит потребление сервером ресурсов: ввода/вывода, памяти и процессора.

  По этой причине, а также чтобы уменьшить риск возникновения ошибок при изменении перечня столбцов таблицы, некоторые администраторы базы данных вообще запрещают применять команду `SELECT *`.

  Разумеется, запрашивать больше данных, чем необходимо, не всегда плохо. Во многих рассмотренных случаях нам говорили, что такой расточительный подход упрощает разработку, так как дает возможность использовать один и тот же код в разных местах. Это разумное соображение, если только вы точно знаете, во что оно обходится с точки зрения производительности. Извлечение лишних данных может быть полезно и для организации некоторых видов кэширования на уровне приложения или получения еще какого-то видимого вам преимущества. Выборка и кэширование полных объектов могут быть предпочтительнее выполнения ряда отдельных запросов, извлекающих части объекта.
- *Повторный выбор одних и тех же данных.* Если не быть осторожными, довольно легко написать код, который повторно получает одни и те же данные с сервера базы данных, выполняя для этого один и тот же запрос. Например, если вы хотите найти URL-адрес аватара пользователя, чтобы отобразить его рядом со списком комментариев, можете запрашивать его для каждого комментария. Или можете кэшировать его, получив первый раз, и затем использовать повторно. Последний подход намного эффективнее.

Не слишком ли много данных анализирует MySQL?

Если вы уверены, что все запросы отбирают лишь необходимые данные, можно поискать те из них, которые для получения результата анализируют слишком много данных. В MySQL простейшими параметрами, с помощью которых можно определить затраты на запрос, являются:

- время отклика;
- количество проанализированных строк;
- количество возвращенных строк.

Ни один из них не является идеальным способом оценки затрат на запрос, но все они дают грубое представление о том, сколько данных MySQL должна прочитать для его выполнения, и приблизительно показывают, насколько быстро работает этот запрос. Все три параметра фиксируются в журнале медленных запросов, так что его просмотр — один из лучших способов выявить запросы, анализирующие слишком много данных.

Время отклика

Не придавайте большого значения времени отклика на запрос. Эй, разве это не противоречит тому, что мы вам говорили? На самом деле нет. Конечно, время отклика имеет значение, но все немного сложнее.

Время отклика состоит из времени обслуживания и времени нахождения в очереди. *Время обслуживания* — это время, необходимое серверу для фактической обработки запроса. *Время нахождения в очереди* — это часть времени отклика, в течение которого сервер не выполняет запрос — он ждет чего-то, например завершения операции ввода/вывода, блокировки строки и т. п. Проблема в том, что вы не можете разбить время отклика на эти составляющие, если не можете их измерять индивидуально, что обычно довольно трудно сделать. Чаще всего вы будете сталкиваться с ожиданием ввода/вывода и блокировки, но стоит учитывать, что подобные ситуации все равно бывают очень разными.

Итак, время отклика соответствует условиям нагрузки. Другие факторы — блокировки подсистемы хранения (табличные и строковые), высокая степень конкурентности и особенности оборудования — тоже могут существенно влиять на время выполнения запроса. Большое время отклика может быть как симптомом, так и причиной проблемы, и не всегда понятно, с чем именно мы столкнулись. Для того чтобы понять, причина это или симптом, можно использовать методику, показанную в разделе «Проблемы одиночного запроса или всего сервера?» главы 3.

Глядя на время отклика, вы должны задать себе вопрос, подходит ли оно для данного запроса. В этой книге нет места для подробного объяснения, но вы можете быстро оценить верхнюю границу (quick upper-bound estimate, QUBE) времени отклика на запрос, используя методы, описанные в книге Тапио Лахденмаки и Майка Лича *Relational Database Index Design and the Optimizers* (издательство Wiley). В двух словах это выглядит так: просмотрите план выполнения запроса и задействованные в нем индексы, определите, сколько может потребоваться последовательных и произвольных операций ввода/вывода, и умножьте их количество на время, необходимое вашему оборудованию для их выполнения. Сложив вычисленное время, вы получите эталон, сравнивая с которым поймете, медленно выполняется запрос или нет.

Количество проанализированных и возвращенных строк

При анализе запросов полезно принимать во внимание, какое количество строк было просмотрено сервером, поскольку это показывает, насколько эффективно запрос находит нужные вам данные. Однако этот параметр неидеален для выявления плохих запросов. Не все обращения к строкам одинаковы. Доступ к коротким строкам занимает меньше времени, а выборка строк из памяти происходит гораздо быстрее, чем их чтение с диска.

В идеале количество проанализированных строк должно совпадать с количеством возвращенных, но на практике так бывает редко. Например, когда в запросе соеди-

няются несколько таблиц, серверу приходится обращаться к нескольким строкам для генерации каждой строки в результирующем наборе. Отношение проанализированных строк к возвращенным обычно мало — скажем, между 1:1 и 10:1, но иногда бывает на несколько порядков больше.

Проанализированные строки и типы доступа

Прикидывая затраты на запрос, учитывайте затраты на поиск одиночной строки в таблице. В MySQL применяется несколько методов поиска и возврата строки. Иногда требуется просмотреть много строк, а иногда удается создать результирующий набор, не анализируя ни одной.

Методы доступа отображаются в столбце `type` результата, возвращаемого командой `EXPLAIN`. Типами доступа могут быть полное сканирование таблицы, сканирование индекса, сканирование диапазона, поиск по уникальному индексу и возврат константы. Каждый следующий быстрее предыдущего, поскольку требует меньшего количества операций чтения. Помнить все типы доступа не обязательно, но очень важно понимать, что означает сканирование таблицы, сканирование индекса, доступ по диапазону и доступ к единственному значению.

Если тип доступа неоптимален, то для решения проблемы лучше всего добавить подходящий индекс. Мы подробно рассматривали вопросы, связанные с индексированием, в предыдущей главе — теперь вы видите, почему индексы так важны для оптимизации запросов. Наличие индекса позволяет MySQL применять гораздо более эффективный метод доступа, при котором приходится анализировать меньше данных.

Рассмотрим для примера простой запрос к демонстрационной базе данных Sakila:

```
mysql> SELECT * FROM sakila.film_actor WHERE film_id = 1;
```

Этот запрос возвращает десять строк, и команда `EXPLAIN` показывает, что MySQL применяет для выполнения запроса тип доступа `ref` по индексу `idx_fk_film_id`:

```
mysql> EXPLAIN SELECT * FROM sakila.film_actor WHERE film_id = 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: idx_fk_film_id
          key: idx_fk_film_id
      key_len: 2
          ref: const
         rows: 10
        Extra:
```

Как показывает команда `EXPLAIN`, MySQL оценила, что придется прочитать всего десять строк. Другими словами, оптимизатор запросов знает, что выбранный тип доступа позволит эффективно выполнить запрос. Что бы случилось, если бы подходящего индекса не было? Тогда MySQL была бы вынуждена воспользоваться менее

эффективным типом доступа. Чтобы убедиться в этом, достаточно удалить индекс и повторить запрос:

```
mysql> ALTER TABLE sakila.film_actor DROP FOREIGN KEY fk_film_actor_film;
mysql> ALTER TABLE sakila.film_actor DROP KEY idx_fk_film_id;
mysql> EXPLAIN SELECT * FROM sakila.film_actor WHERE film_id = 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 5073
        Extra: Using where
```

Как и следовало ожидать, тип доступа поменялся на полное сканирование таблицы (`ALL`), и MySQL определила, что для выполнения запроса придется проанализировать 5073 строки. Слова `Using where` в столбце `Extra` говорят о том, что MySQL использует запрос `WHERE`, чтобы отбросить строки после того, как их считает подсистема хранения.

В целом MySQL может применять запрос `WHERE` тремя способами, которые перечислены далее в порядке от наилучшего к наихудшему.

- Применить указанные условия к операции поиска по индексу, чтобы исключить неподходящие строки. Это происходит на уровне подсистемы хранения.
- Использовать покрывающий индекс (слова `Using index` в столбце `Extra`), чтобы избежать доступа к самим строкам и отфильтровать неподходящие строки после выбора результатов из индекса. Это происходит на уровне сервера, но не требует чтения строк из таблицы.
- Выбрать строки из таблицы, а затем отфильтровать неподходящие (слова `Using where` в столбце `Extra`). Это происходит на уровне сервера, причем сервер перед фильтрацией вынужден считывать строки из таблицы.

Этот пример показывает, насколько важно иметь хорошие индексы. Они позволяют применять при обработке запроса наиболее эффективный тип доступа и анализировать лишь те строки, которые необходимы. Однако добавление индекса не всегда означает, что количество проанализированных и возвращенных строк совпадает. Вот, например, запрос с агрегатной функцией `COUNT()`[1]:

```
mysql> SELECT actor_id, COUNT(*) FROM sakila.film_actor GROUP BY actor_id;
```

Он возвращает всего 200 строк, но для построения результирующего набора сервер должен прочитать тысячи. Индекс не может сократить объем анализируемых данных для подобных запросов.

К сожалению, MySQL ничего не говорит о том, какое количество проанализированных строк использовано для построения результирующего набора, сообщается

[1] Дополнительную информацию см. далее, в разделе «Оптимизация запросов с COUNT()».

лишь, сколько всего строк было проанализировано. Возможно, многие из них оказались отброшены условием `WHERE` и ничего не изменили в результирующем наборе. Если в предыдущем примере удалить индекс по столбцу `sakila.film_actor`, то сервер будет обращаться к каждой строке таблицы, но из-за условия `WHERE` будут отброшены все, кроме десяти. Только оставшиеся десять строк будут использованы для построения результирующего набора. Чтобы понять, к какому числу строк обращается сервер и сколько из них действительно нужны, следует внимательно проанализировать запрос.

Если вы обнаружили, что для получения сравнительно небольшого результирующего набора пришлось проанализировать огромное количество строк, можно попробовать применить некоторые хитрости.

- Воспользуйтесь покрывающими индексами, в которых данные хранятся таким образом, что подсистема хранения вообще не должна извлекать полные строки (мы обсуждали эту тему в предыдущей главе).
- Измените схему. Например, можно использовать сводные таблицы (обсуждали в главе 4).
- Перепишите сложный запрос так, чтобы оптимизатор MySQL мог выполнить его наиболее оптимальным способом (мы обсудим это позже в данной главе).

Способы реструктуризации запросов

При оптимизации проблемных запросов важно найти альтернативные способы получить нужный результат, хотя далеко не всегда это означает, что вы получите точно такой же результирующий набор от MySQL. Иногда удается преобразовать запрос в эквивалентную форму, которая возвращает тот же результат, но с более высокой производительностью. Однако следует подумать и о приведении запроса к виду, дающему *иной* результат, если это позволяет повысить скорость выполнения. Можно даже изменить не только запрос, но и код приложения. В этом разделе мы рассмотрим различные приемы реструктуризации широкого круга запросов и покажем, когда применять каждый из них.

Один сложный или несколько простых запросов?

При создании запросов часто приходится отвечать на важный вопрос: не лучше ли будет разбить сложный запрос на несколько более простых? Традиционный подход при проектировании базы данных состоит в попытках сделать как можно больше работы с помощью наименьшего количества запросов. Исторически такой подход был оправдан из-за высоких затрат на сетевые коммуникации и расходов на разбор и оптимизацию запросов.

Но к MySQL данная рекомендация относится в меньшей степени, поскольку она изначально проектировалась так, чтобы установление и разрыв соединения происходили максимально эффективно, а обработка небольших простых запросов

выполнялась очень быстро. Современные сети также гораздо быстрее, чем прежние, поэтому и сетевые задержки заметно сократились. В зависимости от версии сервера MySQL способна выполнять свыше 100 000 простых запросов в секунду на типичном серверном оборудовании и свыше 2000 запросов в секунду от одиночного клиента в гигабитной сети, поэтому выполнение нескольких запросов может оказаться не такой уж плохой альтернативой.

Хотя передача информации с использованием соединения все же происходит значительно медленнее по сравнению с тем, какой объем находящихся в памяти данных сама MySQL может перебрать в секунду, это количество измеряется миллионами строк. Так что с учетом всех факторов по-прежнему лучше бы ограничиться минимальным количеством запросов, но иногда можно повысить скорость выполнения сложного запроса, разложив его на несколько более простых. Не пугайтесь этого — взвесьте все затраты и выберите ту стратегию, которая уменьшает общий объем работы. Чуть позже мы приведем примеры применения такой методики.

Несмотря на все сказанное, чрезмерно большое количество запросов — одна из наиболее часто допускаемых при проектировании приложений ошибок. Например, в некоторых приложениях выполняется десять запросов, возвращающих по одной строке, вместо одного запроса, отбирающего десять строк. Нам даже встречались приложения, в которых каждый столбец выбирался по отдельности, для чего одна и та же строка запрашивалась многократно!

Разбиение запроса на части

Другой способ уменьшить сложность запроса состоит в применении тактики «разделяй и властвуй», когда выполняется, по существу, один и тот же запрос, но каждый раз из него возвращается меньшее число строк.

Отличный пример — удаление старых данных. В процессе периодической чистки иногда приходится удалять значительные объемы информации, и если делать это одним большим запросом, то возможны блокировка большого числа строк на длительное время, переполнение журналов транзакций, истощение ресурсов, блокировка небольших, не допускающих прерывания запросов. Разбив команду `DELETE` на части и используя запросы среднего размера, мы существенно повысим производительность и уменьшим отставание реплики в случае репликации запроса. Например, вместо следующего монолитного запроса:

```
mysql> DELETE FROM messages WHERE created < DATE_SUB(NOW(),INTERVAL 3 MONTH);
```

вы могли бы выполнить следующие описанные псевдокодом действия:

```
rows_affected = 0
do {
   rows_affected = do_query(
     "DELETE FROM messages WHERE created < DATE_SUB(NOW( ),INTERVAL 3 MONTH)
     LIMIT 10000")
} while rows_affected > 0
```

Обычно удаление 10 000 строк за раз — слишком объемная операция, чтобы быть эффективной, и вместе с тем достаточно короткая, чтобы минимизировать негативное воздействие на сервер[1] (транзакционные подсистемы хранения могут работать эффективнее при меньшем размере транзакции). Кроме того, имеет смысл вставить небольшую паузу между последовательными командами `DELETE`, чтобы распределить нагрузку по времени и не удерживать блокировки слишком долго.

Декомпозиция соединения

Многие высокопроизводительные приложения используют *декомпозицию соединений*. Смысл ее заключается в том, чтобы выполнить несколько однотабличных запросов вместо одного запроса к нескольким объединенным таблицам, а соединение выполнить уже в приложении. Например, следующий запрос:

```
mysql> SELECT * FROM tag
    -> JOIN tag_post ON tag_post.tag_id=tag.id
    -> JOIN post ON tag_post.post_id=post.id
    -> WHERE tag.tag='mysql';
```

можно было бы заменить такими:

```
mysql> SELECT * FROM tag WHERE tag='mysql';
mysql> SELECT * FROM tag_post WHERE tag_id=1234;
mysql> SELECT * FROM post WHERE post.id in (123,456,567,9098,8904);
```

Зачем же, ради всего святого, вы это сделали? На первый взгляд это расточительство, поскольку вы просто увеличили количество запросов, не получив ничего взамен. Однако такая реструктуризация и в самом деле может дать ощутимый выигрыш в производительности.

- Можно более эффективно реализовать кэширование. Во многих приложениях кэшируются объекты, которые полностью соответствуют таблицам. В данном примере, если объект, для которого поле `tag` равно `mysql`, уже кэширован, приложение может пропустить первый запрос. Если выясняется, что в кэше уже есть записи из таблицы `post` с идентификаторами `post_id`, равными 123, 567 или 9098, то соответствующие значения можно исключить из списка `IN()`. Кэш запросов от такой стратегии также выигрывает. Если часто изменяется только одна таблица, то декомпозиция соединения может уменьшить количество удалений записей из кэша.
- Запросы, обращающиеся только к одной таблице, могут иногда уменьшить параллельное выполнение блокировок.
- Соединение результатов на уровне приложения упрощает масштабирование базы данных путем размещения разных таблиц на различных серверах.
- Сами запросы также могут стать более эффективными. В данном примере использование списка `IN()` вместо соединения позволяет MySQL более эффективно

[1] Инструмент pt-archiver пакета Percona Toolkit упрощает реализацию такого рода заданий.

сортировать идентификаторы и более оптимально извлекать строки, чем это было бы возможно в процессе соединения. Подробнее на этом вопросе мы остановимся в дальнейшем.

- Можно избавиться от лишних обращений к строкам. Если соединение производится на уровне приложения, то каждая строка извлекается ровно один раз, тогда как на уровне сервера эта операция, по существу, сводится к денормализации, в ходе которой обращение к одним и тем же данным может выполняться многократно. По той же причине описанная реструктуризация может сократить общий сетевой трафик и потребление памяти.
- В какой-то мере эту методику можно считать ручной реализацией хеш-соединений вместо стандартного алгоритма вложенных циклов, применяемого в MySQL для выполнения соединений. Хеш-соединение может оказаться более эффективным (стратегию соединений MySQL обсудим позже в этой главе).

В результате выполнение объединений на уровне приложения может быть более эффективным в следующих случаях:

- при кэшировании и повторном использовании большого количества данных из более ранних запросов;
- когда данные распределены на нескольких серверах;
- если вместо соединения больших таблиц используются списки `IN()`;
- когда в соединении несколько раз встречается одна и та же таблица.

Основные принципы выполнения запросов

Если вы хотите получать от своего сервера MySQL максимальную производительность, стоит потратить время на изучение того, как MySQL оптимизирует и выполняет запросы. Разобравшись в этом, вы обнаружите, что оптимизация запросов основана на следовании четким принципам и реализована весьма логично.

Другими словами, пришло время повторить то, о чем мы говорили ранее: процесс MySQL следует за выполнением запросов. На рис. 6.1 показано, что происходит, когда вы отправляете запрос к MySQL.

1. Клиент отправляет SQL-команду серверу.
2. Сервер проверяет, есть ли эта команда в кэше запросов. Если да, то возвращается сохраненный результат из кэша, в противном случае выполняется следующий шаг.
3. Сервер осуществляет разбор, предварительную обработку и оптимизацию SQL-команды, преобразуя ее в план выполнения запроса.
4. Подсистема выполнения запросов реализует этот план, обращаясь к подсистеме хранения.
5. Сервер отправляет результат клиенту.

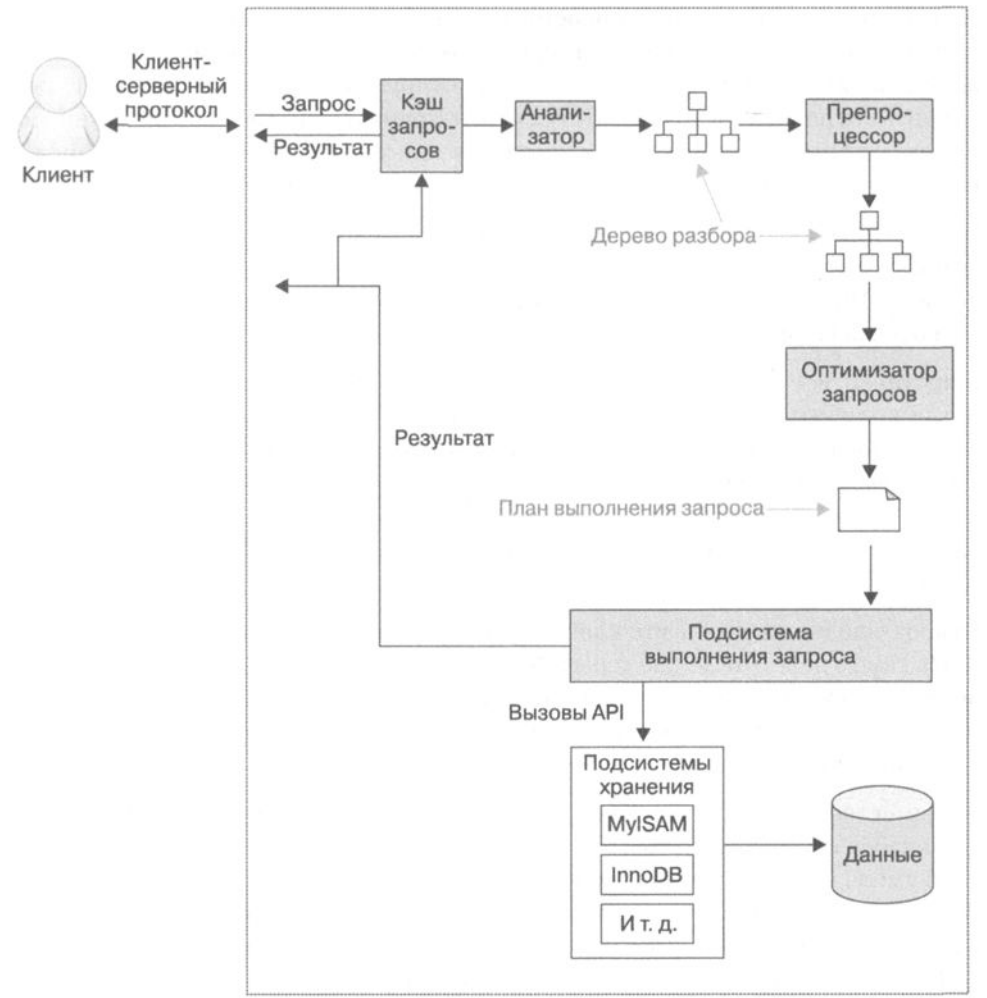


Рис. 6.1. Порядок выполнения запроса

На каждом из этих шагов есть свои тонкости, которые мы обсудим в последующих разделах. Мы также разберем, в каком состоянии запрос находится на каждом из шагов. Особенно сложен и важен для понимания процесс оптимизации. Кроме того, существуют исключения или отдельные случаи (обсудим это в следующей главе).

Клиент-серверный протокол MySQL

Вам не обязательно разбираться во внутренних деталях клиент-серверного протокола MySQL, однако иметь общее представление о том, как он работает, необходимо. Это полудуплексный протокол, то есть в любой момент времени сервер либо отправляет, либо принимает сообщения, но не делает то и другое одновременно. Кроме того, это означает, что невозможно оборвать сообщение на полуслове.

Данный протокол обеспечивает простое и очень быстрое взаимодействие с MySQL, но имеет кое-какие ограничения. Так, в нем отсутствует механизм управления потоком данных: после того как одна сторона отправила сообщение, другая должна получить его целиком и только потом сможет ответить. Это как перекидывание мячика: в каждый момент времени он находится только на одной стороне, невозможно перебросить его сопернику (отправить сообщение), если у вас его нет.

Клиент отправляет запрос в виде одного пакета данных. Поэтому так важна конфигурационная переменная `max_allowed_packet` в случаях, когда встречаются длинные запросы[1]. После отправки запроса мяч уже на другой стороне, клиенту остается только дожидаться результатов.

Напротив, ответ сервера обычно состоит из нескольких пакетов данных. Клиент обязан получить *весь* результирующий набор, отправленный сервером. Нельзя просто выбрать несколько строк и попросить сервер не посылать остальное. Если клиенту все-таки нужны именно первые строки, он может либо дождаться прихода всех отправленных сервером пакетов и отбросить ненужные, либо бесцеремонно разорвать соединение. Оба метода не слишком хороши, поэтому раздел `LIMIT` так важен.

Можно было бы подумать, что клиент, извлекающий строки с сервера, *вытягивает* их. На самом деле это не так: сервер MySQL *выталкивает* строки по мере их создания. Клиент лишь получает вытолкнутые данные, но не может заставить сервер прекратить передачу. Можно сказать, что клиент пьет из пожарного шланга (кстати, это технический термин).

Большинство библиотек для MySQL позволяют либо получить весь результирующий набор и разместить его в памяти, либо выбирать по одной строке. Как правило, по умолчанию весь результирующий набор буферизуется в памяти. Это важно, поскольку до тех пор, пока все строки не будут получены, сервер MySQL не освобождает блокировки и другие ресурсы, потребовавшиеся для выполнения запроса. Запрос будет находиться в состоянии «отправка данных». Если клиентская библиотека извлекает все результаты сразу, то на долю сервера остается меньше работы: он может закончить выполнение запроса и освободить ресурсы максимально быстро.

Большинство клиентских библиотек создают впечатление, что вы получаете данные с сервера, тогда как на самом деле строки выбираются из буфера в памяти самой библиотеки. Как правило, это отлично работает, однако нецелесообразно, если речь идет о гигантских наборах данных, на получение которых уходит много времени, а для хранения требуется много памяти. Можно уменьшить занимаемую память и быстрее приступить к обработке результата, если задать режим работы без буферизации. Недостаток такого подхода заключается в том, что сервер удержи-

[1] Если запрос слишком длинный, сервер отказывается принимать его целиком и возвращает ошибку.

вает блокировки и другие ресурсы на все время, пока приложение взаимодействует с библиотекой[1].

Рассмотрим пример с использованием PHP. Сначала покажем, как обычно отправляется запрос к MySQL из PHP:

```
<?php
$link = mysql_connect('localhost', 'user', 'p4ssword');
$result = mysql_query('SELECT * FROM HUGE_TABLE', $link);
while ( $row = mysql_fetch_array($result) ) {
    // Что-то сделать с результатом
}
?>
```

Код выглядит так, будто строки выбираются по мере необходимости в цикле `while`. Однако на самом деле весь результирующий набор копируется в буфер при вызове функции `mysql_query()`. В цикле `while` мы просто обходим этот буфер. Напротив, в следующем коде результаты не буферизуются, потому что вместо функции `mysql_query()` вызывается `mysql_unbuffered_query()`:

```
<?php
$link = mysql_connect('localhost', 'user', 'p4ssword');
$result = mysql_unbuffered_query('SELECT * FROM HUGE_TABLE', $link);
while ( $row = mysql_fetch_array($result) ) {
    // Что-то сделать с результатом
}
?>
```

В разных языках программирования приняты разные способы подавления буферизации. Например, в драйвере DBD::imysql на Perl необходимо задавать атрибут `mysql_use_result` из клиентской библиотеки C (по умолчанию предполагается наличие атрибута `mysql_store_result`):

```
#!/usr/bin/perl
use DBI;
my $dbh = DBI->connect('DBI:mysql:;host=localhost', 'user', 'p4ssword');
my $sth = $dbh->prepare('SELECT * FROM HUGE_TABLE', { mysql_use_result => 1 });
$sth->execute( ):
while ( my $row = $sth->fetchrow_array( ) ) {
    # Что-то сделать с результатом
}
```

Обратите внимание на то, что в вызове метода `prepare()` атрибут `mysql_use_result` говорит о том, что результат следует использовать, а не буферизовать. Можно было бы задать этот атрибут и на этапе установления соединения, тогда режим буферизации отключался бы для каждой команды:

```
my $dbh = DBI->connect('DBI:mysql:;mysql_use_result=1', 'user', 'p4ssword');
```

Состояния запроса. У каждого соединения, или *потока* MySQL имеется состояние, показывающее, что происходит в данный момент. Существует несколько способов

[1] Это ограничение можно обойти с помощью указания оптимизатору SQL BUFFER_RESULT, которое описывается далее.

узнать состояние, самый простой — воспользоваться командой `SHOW FULL PROCESSLIST` (состояния выводятся в столбце `Command`). На протяжении жизненного цикла запроса состояние меняется много раз, а всего насчитывается несколько десятков состояний. Авторитетным источником информации о состояниях является руководство пользователя по MySQL, мы приведем и поясним лишь некоторые из них.

- `Sleep`. Поток ожидает поступления нового запроса от клиента.
- `Query`. Поток либо выполняет запрос, либо отправляет клиенту результаты.
- `Locked`. Поток ожидает предоставления табличной блокировки на уровне сервера. Блокировки, реализованные подсистемой хранения, например блокировки строк в InnoDB, не вызывают перехода в состояние `Locked`. Это классический симптом блокировки MyISAM, но может встретиться и в других подсистемах хранения, не имеющих блокировки строк.
- `Analyzing` и `statistics`. Поток проверяет статистику, собранную подсистемой хранения, и оптимизирует запрос.
- `Copying to tmp table [on disk]`. Поток обрабатывает запрос и копирует результаты во временную таблицу, скорее всего, для раздела `GROUP BY`, файловой сортировки или удовлетворения запроса, содержащего раздел `UNION`. Если название состояния оканчивается словами `on disk`, значит, MySQL преобразует таблицу в памяти в таблицу на диске.
- `Sorting result`. Поток занят сортировкой результирующего набора.
- `Sending data`. Может означать несколько действий: пересылку данных между различными стадиями обработки запроса, генерацию результирующего набора или возвращение результатов клиенту.

Очень полезно знать хотя бы основные состояния, поскольку это позволяет понять, на чьей стороне мячик. На очень сильно загруженных серверах можно заметить, как редкое или обычно появляющееся на короткое время состояние, например statistics, вдруг начинает занимать много времени. Чаще всего это свидетельствует о возникновении какой-то аномалии. В этом случае вы можете использовать методы, рассмотренные в главе 3, для сбора подробных диагностических данных.

Кэш запросов

Перед тем как приступить к разбору запроса, MySQL проверяет, нет ли его в кэше запросов (если режим кэширования включен). При этом выполняется чувствительный к регистру поиск в хеш-таблице. Если поступивший запрос отличается от хранящегося в кэше хотя бы одним байтом, запросы считаются разными[1] и сервер переходит к следующей стадии обработки запроса.

Если MySQL находит запрос в кэше, то перед тем, как возвратить сохраненный результат, она должна проверить привилегии. Это можно сделать, даже не разбирая

[1] В Percona Server реализована функция, которая до выполнения хеш-поиска удаляет из запросов комментарии. Так кэш запросов используется более эффективно в том случае, когда запросы различаются только текстом комментариев.

запрос, так как вместе с кэшированным запросом MySQL хранит информацию о таблицах. Если с привилегиями все в порядке, MySQL выбирает из кэша ассоциированный с запросом результат и отправляет его клиенту, минуя все остальные стадии. В этом случае запрос не разбирается, не оптимизируется и не выполняется.

Подробнее о кэше запросов рассказывается в главе 7.

Процесс оптимизации запроса

Следующий шаг в жизненном цикле запроса — преобразование SQL-команды в план выполнения, необходимый подсистеме выполнения запросов. Он состоит из нескольких этапов: разбора, предварительной обработки и оптимизации. Ошибки (например, синтаксические) возможны в любой точке этого процесса. Поскольку в нашу задачу не входит строгое документирование внутреннего устройства MySQL, мы можем позволить себе некоторые вольности, например описание шагов по отдельности, хотя ради увеличения эффективности они часто полностью или частично совмещены. Цель — помочь вам разобраться в том, как MySQL выполняет запросы, чтобы вы могли составлять их более оптимально.

Анализатор и препроцессор

Прежде всего *анализатор* MySQL разбивает запрос на лексемы и строит дерево разбора. Для интерпретации и проверки запроса он использует грамматику языка SQL. Проверяет, в частности, все ли лексемы допустимы, располагаются в нужном порядке и нет ли других ошибок, например непарных кавычек.

Затем получившееся дерево разбора передается *препроцессору*, который контролирует дополнительную семантику, не входящую в компетенцию анализатора. К примеру, он проверяет, что указанные таблицы и столбцы существуют, а ссылки на столбцы не допускают неоднозначного толкования.

Далее препроцессор проверяет привилегии. Обычно этот шаг выполняется очень быстро, если только на сервере определено не слишком много привилегий.

Оптимизатор запросов

Теперь, когда дерево разбора тщательно проверено, наступает очередь *оптимизатора* превратить его в план выполнения запроса. Существует множество способов выполнения запроса, все они дают один и тот же результат. Задача оптимизатора — выбрать лучший из них.

В MySQL используется затратный оптимизатор, то есть пытающийся предсказать затраты на реализацию различных планов выполнения и выбрать из них наиболее дешевый. В качестве единицы стоимости ранее принимались затраты на считывание случайной страницы данных размером 4 Кбайт, однако сейчас она стала более сложной и включает такие факторы, как оценочная стоимость выполнения

сравнения WHERE. Чтобы узнать, как оптимизатор оценил запрос, выполните его, а затем посмотрите на сеансовую переменную Last_query_cost:

```
mysql> SELECT SQL_NO_CACHE COUNT(*) FROM sakila.film_actor;
+----------+
| count(*) |
+----------+
|     5462 |
+----------+
mysql> SHOW STATUS LIKE 'Last_query_cost';

+-----------------+-------------+
| Variable_name   | Value       |
+-----------------+-------------+
| Last_query_cost | 1040.599000 |
+-----------------+-------------+
```

Этот результат означает, что, согласно оценке оптимизатора, для выполнения запроса потребуется произвести примерно 1040 случайных чтений страниц данных. Оценка вычисляется на основе различной статистической информации: количества страниц в таблице или в индексе, *кардинальности* (количества различных значений) индекса, длины строк и ключей, распределения ключей. Оптимизатор не учитывает влияния кэширования — предполагается, что любое чтение сводится к операции дискового ввода/вывода.

Оптимизатор не всегда выбирает наилучший план, и тому есть много причин.

- Некорректная статистика. Сервер полагается на статистическую информацию, получаемую от подсистемы хранения, которая может быть как абсолютно верной, так и не имеющей ничего общего с действительностью. Например, подсистема хранения InnoDB из-за своей архитектуры MVCC не ведет точную статистику количества строк в таблице.
- Принятый показатель стоимости не всегда эквивалентен истинной стоимости выполнения запроса. Поэтому, даже когда статистика точна, запрос может оказаться более или менее затратным, чем можно предположить, исходя из оценки MySQL. В некоторых случаях план, предполагающий чтение большего количества страниц, может оказаться менее затратным, потому что, например, чтение с диска производится последовательно или страницы уже кэшированы в памяти. MySQL не понимает, какие страницы находятся в памяти, а какие — на диске, поэтому действительно не знает, сколько запросов ввода/вывода потребуется.
- Представление MySQL о том, что такое оптимально, может расходиться с вашим представлением. Вы, вероятно, хотите получить результат как можно быстрее, однако MySQL на самом деле не пытается ускорить запрос — она стремится минимизировать затраты на его выполнение, а, как мы видели, определение затрат — неточная наука.
- MySQL не принимает в расчет конкурирующие запросы, а они могут повлиять на время обработки оптимизируемого.
- MySQL не всегда выполняет затратную оптимизацию. Иногда она просто следует правилам, например таким: «Если в запросе есть MATCH(), то используется

FULLTEXT-индекс, если таковой существует». Подобное решение будет принято, даже если быстрее было бы воспользоваться другим индексом и неполнотекстовым запросом со словом WHERE.

- Оптимизатор не учитывает затраты на операции, которые ему неподконтрольны, например на выполнение хранимых или определенных пользователем функций.
- Позже мы увидим, что не всегда оптимизатор способен оценить все возможные планы выполнения, поэтому оптимальный план он может просто пропустить.

Оптимизатор запросов MySQL — это очень сложный код, использующий для преобразования запроса в план выполнения множество различных операций. Существует два основных вида оптимизации, которые мы назовем *статической* и *динамической*. *Статическая* оптимизация может быть выполнена на основании только исследования дерева разбора. Например, оптимизатор может преобразовать условие WHERE в эквивалентную форму, применяя алгебраические правила. Статическая оптимизация не зависит от конкретных значений, таких как константы в условии WHERE. Будучи один раз произведенной, статическая оптимизация всегда остается в силе, даже если запрос будет повторно выполнен с другими значениями. Можно считать, что это оптимизация на этапе компиляции.

Динамическая же оптимизация зависит от контекста и может определяться многими факторами, такими как конкретные значения в условии WHERE или количество строк в индексе. Их приходится заново вычислять при каждом выполнении запроса. Можно считать, что это оптимизация на этапе выполнения.

Данное различие важно при выполнении подготовленных операторов и хранимых процедур. MySQL может произвести статическую оптимизацию однократно, но динамическая оптимизация должна заново вычислять при каждом выполнении запроса. Иногда MySQL даже повторно производит оптимизацию во время выполнения запроса[1].

Далее перечислены несколько типов оптимизации, поддерживаемых в MySQL.

- *Изменение порядка соединения.* Таблицы не обязательно надо соединять именно в том порядке, который указан в запросе. Определение наилучшего порядка соединения — важная оптимизация, мы детальнее объясним ее позже в данной главе.
- *Преобразование OUTER JOIN в INNER JOIN.* Оператор OUTER JOIN не обязательно выполнять как внешнее соединение. При определенных условиях, зависящих, например, от раздела WHERE и схемы таблицы, запрос с OUTER JOIN эквивалентен запросу с INNER JOIN. MySQL умеет распознавать и переписывать такие запросы, после чего они могут быть подвергнуты оптимизации типа «изменение порядка соединения».
- *Применение алгебраических правил эквивалентности.* MySQL применяет алгебраические преобразования для упрощения выражений и приведения их к каноническому

[1] Например, план выполнения с проверкой диапазона повторно оценивает индексы для каждой строки в соединении JOIN. Опознать такой план можно по наличию слов range checked for each record в столбце Extra, формируемом командой EXPLAIN. При выборе подобного плана также увеличивается серверная переменная Select_full_range_join server.

виду. Кроме того, она умеет вычислять константные выражения, исключая заведомо невыполнимые и всегда выполняющиеся условия. Например, терм (5=5 AND a>5) приводится к более простому — a>5. Аналогично условие (a<b AND b=c) AND a=5 принимает вид b>5 AND b=c AND a=5. Эти правила очень полезны при написании условных запросов, о чем речь пойдет далее в настоящей главе.

- *Оптимизации COUNT(), MIN() и MAX().* Наличие индексов и сведений о возможности хранения значений NULL в столбцах часто позволяет вообще не вычислять эти выражения. Например, чтобы найти минимальное значение в столбце, который является крайней слева частью индекса, упорядоченного на основе B-дерева, MySQL может просто запросить первую строку из этого индекса. Это можно сделать даже на стадии оптимизации и далее рассматривать полученное значение как константу. Аналогично для поиска максимального значения в индексе, упорядоченном на основе B-дерева, сервер просто считает последнюю строку. При применении такой оптимизации в плане, выведенном командой EXPLAIN, будет присутствовать фраза Select tables optimized away («Некоторые таблицы исключены при оптимизации»). Это означает, что оптимизатор полностью исключил таблицу из плана выполнения, подставив вместо нее константу.

  Подобным образом некоторые подсистемы хранения могут оптимизировать запросы, содержащие COUNT(*) без раздела WHERE (к примеру, MyISAM так хранит данные о количестве строк в таблице).

- *Вычисление и свертка константных выражений.* Если MySQL обнаруживает, что выражение можно свернуть в константу, то делает это на стадии оптимизации. Например, определенную пользователем переменную можно преобразовать в константу, если она не изменяется в запросе. Другим примером могут служить арифметические выражения.

  Как ни странно, даже такие вещи, которые вы, скорее всего, назвали бы запросом, можно свернуть в константу во время оптимизации. Например, вычисление функции MIN() по индексу. Этот пример можно даже расширить до поиска констант по первичному ключу или по уникальному индексу. Если в разделе WHERE встречается константное условие для такого индекса, то оптимизатор знает, что MySQL может найти значение в самом начале выполнения запроса. Впоследствии найденное значение можно трактовать как константу. Приведем пример:

```
mysql> EXPLAIN SELECT film.film_id, film_actor.actor_id
    -> FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    -> WHERE film.film_id = 1;
+----+-------------+------------+-------+---------------+-------+------+
| id | select_type | table      | type  | key           | ref   | rows |
+----+-------------+------------+-------+---------------+-------+------+
|  1 | SIMPLE      | film       | const | PRIMARY       | const |    1 |
|  1 | SIMPLE      | film_actor | ref   | idx_fk_film_id | const |   10 |
+----+-------------+------------+-------+---------------+-------+------+
```

MySQL выполняет этот запрос в два этапа, о чем свидетельствуют две строки в выведенной таблице. На первом этапе она находит нужную строку в таблице film. Оптимизатор MySQL знает, что такая строка единственная, поскольку столбец film_id — это первичный ключ, а оптимизатор уже на стадии оптимизации

запроса уточнил количество найденных строк. Так как оптимизатору известна точная величина (значение в разделе `WHERE`), которая будет возвращена в результате поиска, то в столбце `ref` для этой таблицы стоит `const`.

На втором этапе MySQL считает столбец `film_id` из строки, найденной на первом шаге, известной величиной. Она может так поступить, потому что в момент перехода к второму шагу оптимизатор уже знает все значения, определенные ранее. Обратите внимание на то, что тип `ref` для таблицы `film_actor` равен `const` точно так же, как и для таблицы `film`.

Константные условия могут применяться также вследствие распространения константности значения от одного места к другому при наличии разделов `WHERE`, `USING` или `ON` с ограничением типа «равно». В приведенном ранее примере фраза `USING` гарантирует, что значение `film_id` будет одинаково на протяжении всего запроса — оно должно быть равно константе, заданной в разделе `WHERE`.

- *Покрывающие индексы.* Если индекс содержит все необходимые запросу столбцы, то MySQL может воспользоваться им, вообще не читая данные таблицы. Мы подробно рассматривали покрывающие индексы в предыдущей главе.
- *Оптимизация подзапросов.* MySQL умеет преобразовывать некоторые виды подзапросов в более эффективные альтернативные формы, сводя их к поиску по индексу.
- *Раннее завершение.* MySQL может прекратить обработку запроса (или какой-то шаг обработки), как только поймет, что этот запрос или шаг полностью выполнен. Очевидный пример — раздел `LIMIT`, но есть и еще несколько случаев раннего завершения. Например, встретив заведомо невыполнимое условие, MySQL может прекратить обработку всего запроса. Рассмотрим следующий пример:

```
mysql> EXPLAIN SELECT film.film_id FROM sakila.film WHERE film_id = -1;
+----+...+-------------------------------------------------------+
| id |...| Extra                                                 |
+----+...+-------------------------------------------------------+
|  1 |...| Impossible WHERE noticed after reading const tables |
+----+...+-------------------------------------------------------+
```

Этот запрос остановлен на шаге оптимизации, но в некоторых других случаях MySQL также прерывает выполнение рано. Сервер может применить такую оптимизацию, когда подсистема выполнения приходит к выводу, что нужно извлекать только различающиеся значения либо остановиться, если значения не существует. Например, следующий запрос находит все фильмы, в которых нет ни одного актера[1]:

```
mysql> SELECT film.film_id
    -> FROM sakila.film
    ->    LEFT OUTER JOIN sakila.film_actor USING(film_id)
    -> WHERE film_actor.film_id IS NULL;
```

[1] Мы согласны, что фильм без актеров выглядит странно, но в демонстрационной базе данных Sakila такой фильм есть. Он называется Slacker Liaisons и аннотирован как «стремительно развивающаяся история о мошеннике и студенте, которые должны встретиться с крокодилом в Древнем Китае».

Запрос исключает все фильмы, в которых есть хотя бы один актер. В фильме может быть задействовано много актеров, но, обнаружив первого, сервер прекращает обработку текущего фильма и переходит к следующему, поскольку знает, что условию `WHERE` такой фильм заведомо не удовлетворяет. Похожую оптимизацию «различается/не существует» можно применить к некоторым типам запросов, включающих операторы `DISTINCT`, `NOT EXISTS()` и `LEFT JOIN`.

- *Распространение равенства.* MySQL распознает ситуации, когда в некотором запросе два столбца должны быть равны, например в условии `JOIN`, и распространяет условие `WHERE` на эквивалентные столбцы. В частности, из запроса:

```
mysql> SELECT film.film_id
    -> FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    -> WHERE film.film_id > 500;
```

MySQL понимает, что условие `WHERE` применяется не только к таблице `film`, но и к таблице `film_actor`, поскольку в силу наличия раздела `USING` оба столбца должны совпадать.

Если вы привыкли к другой СУБД, в которой такая оптимизация не реализована, то вам, вероятно, рекомендовали «помочь оптимизатору», самостоятельно задав в разделе `WHERE` условия для обеих таблиц, например:

```
... WHERE film.film_id > 500 AND film_actor.film_id > 500
```

В MySQL это не обязательно и лишь усложняет поддержку запросов.

- *Сравнение по списку IN().* Во многих базах данных оператор `IN()` является не более чем синонимом нескольких условий OR, поскольку логически они эквивалентны. Но не в MySQL, которая сортирует значения в списке `IN()` и применяет для работы с ним быстрый двоичный поиск. Вычислительная сложность при этом составляет `O(log n)`, где `n` — размер списка, тогда как сложность эквивалентной последовательности условий OR равна `O(n)` (то есть гораздо медленнее для больших списков).

Приведенный перечень, естественно, неполон, так как MySQL умеет применять гораздо больше оптимизаций, чем поместилось бы во всей этой главе, но представление о сложности и развитых логических возможностях оптимизатора вы все же получили. Главная мысль, которую следует вынести из этого обсуждения: *не пытайтесь перехитрить оптимизатор.* В итоге вы просто потерпите неудачу или сделаете свои запросы чрезмерно запутанными и сложными для сопровождения, не получив ни малейшей выгоды. Позвольте оптимизатору заниматься своим делом.

Разумеется, каким бы умным ни был оптимизатор, случается, что он не находит наилучшего результата. Иногда вы знаете о своих данных что-то такое, о чем оптимизатору неизвестно, например некое условие, которое гарантированно истинно вследствие логики приложения. Кроме того, иногда оптимизатор не наделен необходимой функциональностью, например хеш-индексами, а временами алгоритм оценки стоимости выбирает план, оказывающийся более затратным, чем возможная альтернатива.

Если вы знаете, что оптимизатор дает плохой результат, и вам известно, почему так произошло, можете ему помочь. В данном случае можно включить в запрос подсказку, переписать запрос, перепроектировать схему или добавить индексы.

Статистика по таблицам и индексам

Вспомните архитектурные уровни сервера MySQL, показанные на рис. 1.1. На уровне сервера, где расположен оптимизатор запросов, отсутствует статистика по данным и индексам. Задание обеспечить ее возлагается на подсистемы хранения, поскольку каждая из них может собирать различную статистическую информацию (и хранить ее разными способами). Некоторые подсистемы, например Archive, вообще не собирают статистику!

Поскольку сервер не содержит статистики, оптимизатор запросов MySQL должен обращаться к подсистемам хранения за статистическими данными по участвующим в запросе таблицам. Подсистема может предоставить такую информацию, как количество страниц в таблице или индексе, кардинальность таблиц и индексов, длина строк и ключей, данные о распределении ключей. На основе этих сведений оптимизатор выбирает наилучший план выполнения. В последующих разделах мы увидим, как статистика влияет на решения оптимизатора.

Стратегия выполнения соединений в MySQL

В MySQL термин «соединение» используется шире, чем вы, возможно, привыкли. Под соединениями понимаются все запросы, а не только те, что сопоставляют строки из двух таблиц. Еще раз подчеркнем: имеются в виду все без исключения запросы, в том числе подзапросы и даже выборка `SELECT` из одной таблицы. Следовательно, очень важно понимать, как в MySQL выполняется операция соединения.

Рассмотрим, к примеру, запрос `UNION`. MySQL реализует операцию `UNION` в виде последовательности отдельных запросов, результаты которых сохраняются во временной таблице, а затем снова читаются из нее. Каждый запрос представляет собой соединение в терминологии MySQL, и им же является операция чтения из результирующей временной таблицы.

В текущей версии стратегия соединения в MySQL проста: все соединения реализуются алгоритмом вложенных циклов. Это означает, что MySQL в цикле перебирает строки из одной таблицы, а затем во вложенном цикле ищет соответствующие строки в следующей. Это продолжается до тех пор, пока не будут найдены соответствующие строки во всех таблицах. После этого строится и возвращается строка, составленная из перечисленных в списке `SELECT` столбцов. Далее MySQL пытается найти следующую строку в последней таблице. Если такой не оказывается, то происходит возврат на одну таблицу назад и делается попытка найти дополнительные строки в ней. MySQL продолжает выполнять возвраты, пока в какой-то таблице не обнаружится подходящая строка, после чего ищет соответствующую ей строку в следующей таблице и т. д.[1]

[1] Далее мы покажем, что на самом деле выполнение запроса не такое простое дело: существует немало оптимизаций, усложняющих алгоритм.

Этот процесс, состоящий из поиска строк, проверки следующей таблицы и возврата, в плане выполнения можно представить в виде последовательности вложенных циклов. Отсюда и название — *соединение методом вложенных циклов*.

В качестве примера рассмотрим простой запрос:

```
mysql> SELECT tbl1.col1, tbl2.col2
    -> FROM tbl1 INNER JOIN tbl2 USING(col3)
    -> WHERE tbl1.col1 IN(5,6);
```

Предположим, что MySQL решает соединить таблицы в порядке, указанном в запросе. Приведенный далее псевдокод показывает возможный алгоритм выполнения запроса:

```
outer_iter = iterator over tbl1 where col1 IN(5,6)
outer_row  = outer_iter.next
while outer_row
    inner_iter = iterator over tbl2 where col3 = outer_row.col3
    inner_row  = inner_iter.next
    while inner_row
       output [ outer_row.col1, inner_row.col2 ]
       inner_row = inner_iter.next
    end
    outer_row = outer_iter.next
 end
```

Этот план выполнения с тем же успехом можно применить к запросу, в котором участвует всего одна таблица. Именно поэтому запросы к одной таблице считаются соединениями: такое однотабличное соединение представляет собой базовую операцию, на основе которой конструируются более сложные соединения. Данный алгоритм распространяется и на соединения `OUTER JOIN`. Например, слегка изменим предыдущий запрос:

```
mysql> SELECT tbl1.col1, tbl2.col2
    -> FROM tbl1 LEFT OUTER JOIN tbl2 USING(col3)
    -> WHERE tbl1.col1 IN(5,6);
```

Вот как выглядит соответствующий ему псевдокод (изменения выделены полужирным шрифтом):

```
outer_iter = iterator over tbl1 where col1 IN(5,6)
outer_row  = outer_iter.next
while outer_row
    inner_iter = iterator over tbl2 where col3 = outer_row.col3
    inner_row  = inner_iter.next
    if inner_row
       while inner_row
          output [ outer_row.col1, inner_row.col2 ]
          inner_row = inner_iter.next
       end
    else
       output [ outer_row.col1, NULL ]
    end
    outer_row = outer_iter.next
end
```

Еще одним способом визуализации плана выполнения запроса является так называемая *диаграмма дорожек для плавания*. На рис. 6.2 показана такая диаграмма для исходного запроса с `INNER JOIN`. Читать ее следует слева направо и сверху вниз.

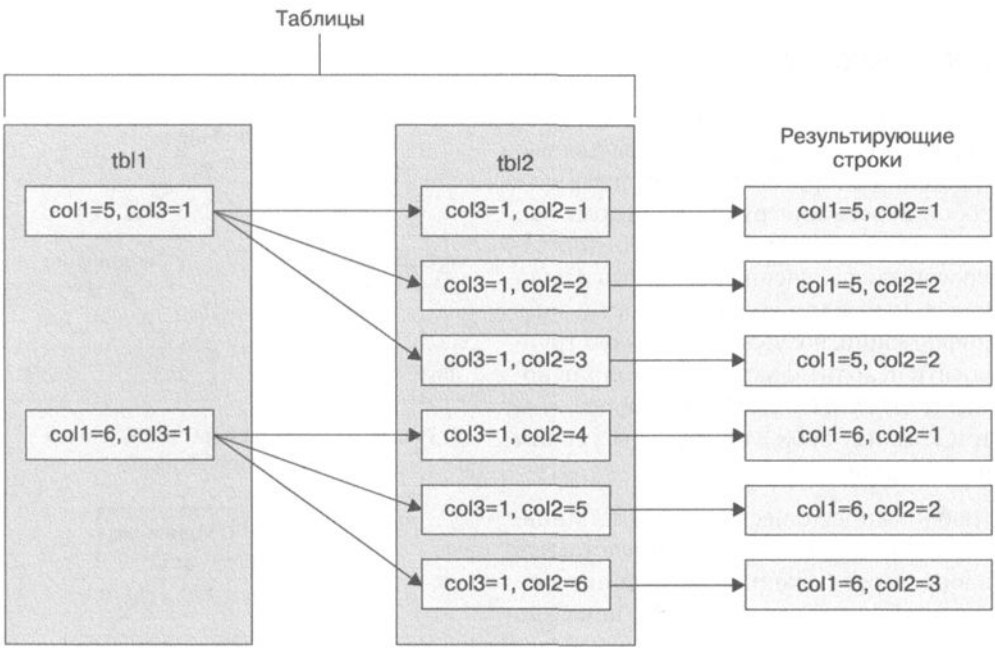


Рис. 6.2. Диаграмма дорожек для плавания, иллюстрирующая выбор строк при обработке соединения

MySQL выполняет запросы любого вида практически одинаково. Например, подзапрос в разделе `FROM` выполняется первым, его результаты сохраняются во временной таблице[1], которая затем трактуется как самая обычная таблица (отсюда и название — *производная таблица*). При обработке запросов с `UNION` тоже используются временные таблицы, а запросы с `RIGHT OUTER JOIN` преобразуются в эквивалентную форму с `LEFT OUTER JOIN`. Короче говоря, MySQL приводит запросы любого вида к этому плану выполнения[2].

Но таким способом можно выполнить не все допустимые SQL-запросы. Например, запрос `FULL OUTER JOIN` не удастся выполнить методом вложенных циклов с возвратами, если обнаружится таблица, не содержащая подходящих строк, поскольку

[1] Во временных таблицах не строятся индексы, и об этом следует помнить при написании сложных соединений с результатами подзапросов в разделе FROM. То же самое относится и к запросам с UNION.

[2] В MySQL 5.6 и MariaDB произошли существенные изменения, что позволило задействовать более сложные пути выполнения.

алгоритм может начать работу именно с нее. Вот поэтому MySQL не поддерживает оператор `FULL OUTER JOIN`. Есть и другие запросы, которые хотя и можно выполнить методом вложенных циклов, но это окажется крайне неэффективным. Мы рассмотрим такие примеры в дальнейшем.

План выполнения

В отличие от многих других СУРБД MySQL не генерирует байт-код для выполнения запроса. План представляет собой дерево инструкций, которые выполняются подсистемой выполнения запросов для получения результата. Окончательный план содержит достаточно информации, позволяющей реконструировать исходный запрос. Выполнив команду `EXPLAIN EXTENDED` для запроса, а затем команду `SHOW WARNINGS`, мы увидим реконструированный запрос[1].

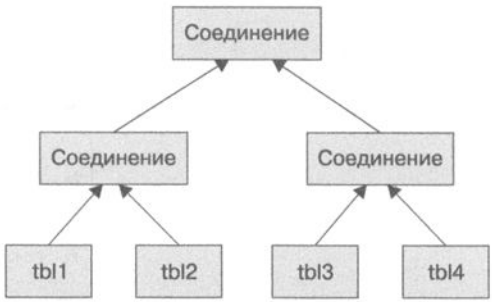


Рис. 6.3. Один из способов соединения нескольких таблиц

Любой запрос с несколькими таблицами концептуально можно представить в виде дерева. Например, соединить четыре таблицы можно так, как показано на рис. 6.3.

Специалист по информатике назовет это *сбалансированным деревом*. Но MySQL обрабатывает запрос иначе. В предыдущем разделе мы рассказали, что MySQL всегда начинает с какой-то одной таблицы и ищет соответствующие строки в следующей. Поэтому планы выполнения запросов, формируемые MySQL, всегда имеют вид *левоглубинного дерева* (left-deep tree) (рис. 6.4).

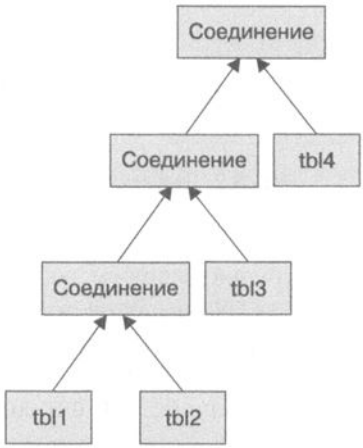


Рис. 6.4. Соединение нескольких таблиц MySQL

Оптимизатор соединений

Важнейшая часть оптимизатора запросов в MySQL — это *оптимизатор соединений*, который определяет наилучший порядок выполнения многотабличных запросов. Часто для получения одинаковых результатов таблицы можно соединять в разном

[1] Сервер генерирует выходную информацию, исходя из плана выполнения. Поэтому выведенный запрос будет иметь ту же семантику, что и исходный, но не обязательно тот же самый текст.

порядке. Оптимизатор соединений вычисляет затраты на реализацию различных планов и пытается выбрать наименее затратный.

Приведем пример запроса, в котором таблицы можно соединить в разном порядке с одним и тем же результатом:

```
mysql> SELECT film.film_id, film.title, film.release_year, actor.actor_id,
    ->    actor.first_name, actor.last_name
    ->    FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    ->    INNER JOIN sakila.actor USING(actor_id);
```

Скорее всего, вы сможете придумать несколько планов выполнения. Например, MySQL могла бы, начав с таблицы `film`, воспользоваться индексом таблицы `film_actor` по полю `film_id` для поиска значений `actor_id`, а затем искать строки по первичному ключу таблицы `actor`. Пользователи Oracle, возможно, перефразируют это так: «Таблица `film` — это ведущая таблица для таблицы `film_actor`, которая является ведущей для таблицы `actor`». Это должно быть эффективно, не правда ли? Однако посмотрим на вывод команды `EXPLAIN`, объясняющей, как MySQL в действительности собирается выполнять этот запрос:

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 200
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: PRIMARY,idx_fk_film_id
          key: PRIMARY
      key_len: 2
          ref: sakila.actor.actor_id
         rows: 1
        Extra: Using index
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: eq_ref possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.film_actor.film_id
         rows: 1
        Extra:
```

Этот план существенно отличается от предложенного ранее. MySQL хочет начать с таблицы `actor` (мы знаем это, потому что она идет первой в списке, выданном

командой `EXPLAIN`) и двигаться в обратном порядке. Действительно ли это более эффективно? Попробуем разобраться. Ключевое слово `STRAIGHT_JOIN` заставляет сервер выполнять соединение в той последовательности, которая указана в запросе. Вот что говорит `EXPLAIN` о таком модифицированном запросе:

```
mysql> EXPLAIN SELECT STRAIGHT_JOIN film.film_id...\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 951
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: PRIMARY,idx_fk_film_id
          key: idx_fk_film_id
      key_len: 2
          ref: sakila.film.film_id
         rows: 1
        Extra: Using index
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: eq_ref possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.film_actor.actor_id
         rows: 1
        Extra:
```

Теперь мы видим, почему MySQL предпочла обратный порядок соединения: это позволяет исследовать меньше строк в первой таблице[1]. В обоих случаях во второй и третьей таблицах можно вести быстрый поиск по индексу. Разница лишь в том, сколько таких операций поиска придется выполнить.

- Если начать с таблицы `film`, то потребуется 951 обращение к таблицам `film_actor` и `actor`, по одному для каждой строки в первой таблице.
- Если же сначала искать в таблице `actor`, то для последующих таблиц придется выполнить всего 200 операций поиска по индексу.

[1] Если строго, то MySQL не стремится минимизировать количество прочитанных строк. Она пытается оптимизировать количество считываемых страниц. Однако счетчик строк часто дает грубое представление о стоимости запроса.

Другими словами, измененный порядок соединений сокращает количество возвратов и повторных операций чтения. Чтобы еще раз убедиться в правильности выбранного оптимизатором решения, мы выполнили оба варианта запроса и посмотрели на значение переменной `Last_query_cost`. Затраты на запрос с измененным порядком соединения составили 241, а на запрос с навязанным нами порядком — 1154.

Это простой пример того, как оптимизатор соединений в MySQL может добиваться более низких затрат, изменяя порядок выполнения запросов. Изменение порядка соединений обычно оказывается весьма эффективной оптимизацией. Но иногда она не дает оптимального плана, и в этих случаях вы можете употребить ключевое слово `STRAIGHT_JOIN` и написать запрос в том порядке, который вам представляется наилучшим. Однако такие случаи редки — чаще всего оптимизатор оказывается эффективнее человека.

Оптимизатор соединений пытается построить дерево плана выполнения запроса с минимальными затратами. Если возможно, он исследует все потенциальные перестановки поддеревьев.

К сожалению, для полного исследования соединения n таблиц нужно перебрать n-факториал возможных перестановок. Это называется *пространством поиска* всех возможных планов выполнения. Его размер растет очень быстро — соединение десяти таблиц можно осуществить 3 628 800 способами! Если пространство поиска оказывается слишком большим, то для оптимизации запроса потребуется чересчур много времени, поэтому сервер отказывается от полного анализа. Если количество соединяемых таблиц превышает значение параметра `optimizer_search_depth` (которое вы при желании можете изменить), то сервер применяет сокращенные алгоритмы, например жадный поиск.

Для ускорения стадии оптимизации в MySQL реализовано множество эвристических приемов, выработанных за годы исследований и экспериментов. Это может быть полезно, но также означает, что иногда (в редких случаях) MySQL может пропустить оптимальный план и выбрать менее удачный, поскольку она не пыталась исследовать все возможности.

Иногда запросы нельзя выполнить в другом порядке, тогда, воспользовавшись этим фактом, оптимизатор соединений может сократить пространство поиска, исключив из него некоторые перестановки. Хорошим примером могут служить запросы с `LEFT JOIN` и коррелированные подзапросы (к подзапросам мы еще вернемся). Объясняется это тем, что результаты для одной таблицы зависят от данных, извлеченных из другой. Наличие таких зависимостей позволяет оптимизатору соединений сократить пространство поиска с помощью исключения выбора.

Оптимизации сортировки

Сортировка результатов может оказаться весьма затратной операцией, поэтому часто производительность можно повысить, исключив ее или уменьшив количество сортируемых строк.

В главе 3 мы показали, как индексы могут быть использованы для сортировки. Если MySQL не может найти подходящего индекса, то ей приходится сортировать строки самостоятельно. Это можно сделать в памяти или на диске, но сама процедура всегда называется *файловой сортировкой*, даже если файл в действительности не используется.

Если обрабатываемые данные умещаются в буфер, то MySQL может выполнить сортировку целиком в памяти, применяя алгоритм *быстрой сортировки*. В противном случае сортировка реализуется на диске поблочно. Каждый блок обрабатывается методом быстрой сортировки, а затем отсортированные блоки сливаются.

Есть два алгоритма файловой сортировки.

- *Двухпроходный (старый).* Читает указатели на строки и столбцы, упомянутые в разделе `ORDER BY`, сортирует их, затем проходит по отсортированному списку и снова читает исходные строки, чтобы вывести результат.

  Двухпроходный алгоритм может быть довольно затратным, поскольку строки из таблицы прочитываются дважды и повторное чтение вызывает много произвольных операций ввода/вывода. Особенно накладно это в случае таблиц типа MyISAM, так как для выборки каждой строки необходимо вызывать операционную систему, поскольку MyISAM полагается на нее в вопросах кэширования данных. В то же время при такой сортировке хранится небольшой объем данных, поэтому если все сортируемые строки уже находятся в памяти, то, возможно, менее затратно хранить меньше данных и перечитывать строки для генерации окончательного результата.
- *Однопроходный (новый).* Читает все необходимые запросу столбцы, сортирует строки по столбцам, упомянутым в разделе `ORDER BY`, проходит по отсортированному списку и выводит заданные столбцы.

  Этот алгоритм реализуется, начиная с версии MySQL 4.1. Он может быть намного эффективнее старого, особенно на наборах данных с большой нагрузкой на ввод/вывод. Однопроходный алгоритм не читает строки из таблицы дважды, а произвольный ввод/вывод в нем заменяется последовательным чтением. Но при этом необходимо больше памяти, так как для каждой строки приходится хранить все запрошенные столбцы, а не только те, по которым производится сортировка. Следовательно, в буфер сортировки поместится меньше строк и надо будет выполнить больше циклов слияния.

Трудно сказать, какой алгоритм более эффективен, поскольку есть подходящие и неподходящие случаи для применения каждого из них. MySQL использует новый алгоритм, если общий размер всех столбцов, необходимых для запроса, плюс столбцы `ORDER BY` не больше чем `max_length_for_sort_data` байтов. Таким образом, вы можете использовать этот параметр для выбора используемого алгоритма. Подробнее об этом см. в подразделе «Оптимизация файловой сортировки» в главе 8.

При файловой сортировке серверу MySQL может потребоваться гораздо больше места на диске для хранения временных файлов, чем вы ожидаете, так как каждому сортируемому кортежу выделяется запись фиксированной длины. Ее размера долж-

но хватать для того, чтобы сохранить максимально длинный кортеж, в котором для столбцов типа `VARCHAR` зарезервировано место в соответствии с указанным в схеме размером. Кроме того, для кодировки UTF-8 MySQL выделяет на каждый символ 3 байта. Так что нам доводилось встречать плохо оптимизированные схемы, в которых размер временного файла сортировки во много раз превышал размер исходной таблицы на диске.

При выполнении соединения MySQL может производить файловую сортировку на двух стадиях реализации запроса. Если в разделе `ORDER BY` упомянуты только столбцы из первой (в порядке соединения) таблицы, то MySQL может провести ее файловую сортировку, а затем приступить к соединению. В таком случае команда `EXPLAIN` поместит в столбец `Extra` фразу `Using filesort`. Во всех других обстоятельствах — например, сортируемая таблица не первая по порядку или раздел `ORDER BY` содержит столбцы более чем из одной таблицы — MySQL должна хранить результаты запроса во временной таблице, а выполнять файловую сортировку временной таблицы после завершения соединения. В таком случае команда `EXPLAIN` поместит в столбец `Extra` фразу `Using temporary; Using filesort`. Если задано ключевое слово `LIMIT`, то оно применяется после сортировки, поэтому временная таблица и файловая сортировка могут быть очень велики.

В версии MySQL 5.6 произошли значительные изменения в алгоритме сортировки, когда необходимо выбрать лишь подмножество строк, например при использовании ключевого слова `LIMIT`. Иногда вместо того, чтобы сортировать весь набор результатов и затем возвращать его часть, MySQL 5.6 позволяет отбрасывать ненужные строки перед сортировкой.

Подсистема выполнения запросов

На стадиях разбора и оптимизации вырабатывается план выполнения запроса, который применяет подсистема выполнения. План представляет собой структуру данных, а не исполняемый байт-код, как во многих других базах данных.

Как правило, стадия выполнения гораздо проще, чем стадия оптимизации: MySQL просто следует инструкциям, содержащимся в плане выполнения запроса. Для многих указанных в плане операций нужно вызывать методы, реализованные в подсистеме хранения, интерфейс с этой подсистемой еще называют *API обработчика*. Каждая таблица в запросе представлена экземпляром обработчика. Если таблица встречается в запросе трижды, то сервер сформирует три экземпляра обработчика. Хотя раньше мы об этом не упоминали, фактически MySQL создает экземпляры обработчика на ранних шагах стадии оптимизации. Оптимизатор использует их для получения информации о таблицах, например о названиях столбцов и статистике по индексам.

В интерфейсе подсистемы хранения определено множество функций, но для выполнения большинства запросов хватает примерно десятка основных операций-кирпичиков. Например, имеются операции для чтения первой и следующей строки в индексе. Этого вполне достаточно для запроса, в котором просматривается индекс.

Благодаря такому упрощенному подходу к выполнению запроса и стало возможным использование архитектуры подсистемы хранения в MySQL, но это, в свою очередь, накладывает определенные ограничения на оптимизатор, которые мы рассмотрели ранее.

Не все, что включено в план выполнения запроса, — это операции обработчика. Например, сервер управляет табличными блокировками. Обработчик может реализовать собственную схему блокировки на нижнем уровне строк, как делает, например, InnoDB со строковыми блокировками (однако она не заменяет реализацию блокировок внутри сервера). В главе 1 отмечалось, что на сервере реализованы функции, общие для всех подсистем хранения, например функции работы с датой и временем, представления и триггеры.

Для выполнения запроса сервер просто повторяет инструкции до тех пор, пока не будут проанализированы все строки.

Возврат результатов клиенту

Последняя стадия выполнения запроса — отправка ответа клиенту. Даже запросы, которые не возвращают результирующий набор, все равно посылают клиенту информацию о результате обработки, например о том, сколько строк было изменено.

Если запрос можно кэшировать, то на этой стадии MySQL поместит результаты в кэш запросов.

Сервер генерирует и отправляет данные с определенным шагом. Давайте вернемся к рассмотренному ранее алгоритму соединения нескольких таблиц. Как только MySQL обработает последнюю таблицу и успешно сгенерирует очередную строку, она может и должна отправить ее клиенту. У такого решения есть два достоинства: серверу не обязательно хранить строку в памяти, а клиент начинает получать результаты настолько быстро, насколько это вообще возможно[1].

Каждая строка в результирующем наборе отправляется в отдельном пакете по протоколу клиента — сервера MySQL, хотя пакеты протоколов могут быть буферизованы и отправлены вместе на уровне протокола TCP.

Ограничения оптимизатора MySQL

Подход MySQL к выполнению запросов по принципу «всякий запрос — это соединение методом вложенных циклов» неидеален для оптимизации всех запросов. К счастью, есть не так уж много случаев, с которыми оптимизатор MySQL справляется

[1] При желании это поведение можно изменить, например указав SQL_BUFFER_RESULT. См. раздел «Подсказки оптимизатору запросов» далее в этой главе.

заведомо плохо, и обычно удается переписать такие запросы более эффективно. Более того, после выпуска MySQL 5.6 устранит многие ограничения MySQL и сделает множество запросов намного более быстрыми.

Коррелированные подзапросы

Иногда MySQL очень плохо оптимизирует подзапросы. Самый неприятный случай — подзапросы в операторе `IN()` в разделе `WHERE`. Например, найдем в демонстрационной базе Sakila все фильмы, в которых играла Пенелопа Гинесс (`actor_id=1`). Вполне естественным кажется написание такого подзапроса:

```
mysql> SELECT * FROM sakila.film
    -> WHERE film_id IN(
    ->     SELECT film_id FROM sakila.film_actor WHERE actor_id = 1);
```

Хочется думать, что MySQL будет выполнять этот запрос изнутри наружу, то есть сначала найдет список значений `film_id` с заданным `actor_ id`, а затем подставит их в список `IN()`. Ранее мы отмечали, что в общем случае списки `IN()` обрабатываются очень быстро, поэтому можно ожидать, что запрос будет оптимизирован следующим образом:

```
- SELECT GROUP_CONCAT(film_id) FROM sakila.film_actor WHERE actor_id = 1;
-- Result: 1,23,25,106,140,166,277,361,438,499,506,509,605,635,749,832,939, 970,980
SELECT * FROM sakila.film
WHERE film_id
IN(1,23,25,106,140,166,277,361,438,499,506,509,605,635,749,83 2,939,970,980);
```

К сожалению, на деле произойдет прямо противоположное. MySQL попытается помочь подзапросу, перенеся в него корреляцию из внешней таблицы, так как полагает, что так подзапрос будет искать строки эффективнее. В результате запрос переписывается следующим образом:

```
SELECT * FROM sakila.film
WHERE EXISTS (
    SELECT * FROM sakila.film_actor WHERE actor_id = 1
    AND film_actor.film_id = film.film_id);
```

Теперь подзапросу требуется значение поля `film_id` из внешней таблицы `film`, поэтому первым его выполнить не получится. `EXPLAIN` показывает в качестве типа запроса `DEPENDENT SUBQUERY` (вы можете точно узнать, как был переписан запрос, при помощи команды `EXPLAIN EXTENDED`):

```
mysql> EXPLAIN SELECT * FROM sakila.film ...;
+----+--------------------+------------+--------+-----------------------+
| id | select_type        | table      | type   | possible_keys         |
+----+--------------------+------------+--------+-----------------------+
|  1 | PRIMARY            | film       | ALL    | NULL                  |
|  2 | DEPENDENT SUBQUERY | film_actor | eq_ref | PRIMARY,idx_fk_film_id |
+----+--------------------+------------+--------+-----------------------+
```

Согласно результатам `EXPLAIN`, MySQL производит полное сканирование таблицы `film` и выполняет подзапрос для каждой найденной строки. Если таблица небольшая, это не приведет к заметному падению производительности, но когда внешняя таблица

велика, замедление окажется весьма существенным. К счастью, такой запрос легко переписать с использованием оператора JOIN:

```
mysql> SELECT film.* FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    -> WHERE actor_id = 1;
```

Еще один неплохой вариант оптимизации — вручную сгенерировать список IN(), выполнив вместо подзапроса отдельный запрос с функцией GROUP_CONCAT(). Иногда это оказывается быстрее, чем JOIN. И наконец, даже если подзапросы IN() работают плохо во многих случаях, EXISTS () или подзапросы равенства иногда работают намного лучше. Приведем еще один способ переписывания подзапроса IN():

```
mysql> SELECT * FROM sakila.film
    -> WHERE EXISTS(
    ->    SELECT * FROM sakila.film_actor WHERE actor_id = 1
    ->       AND film_actor.film_id = film.film_id);
```

Ограничения оптимизатора, которые мы обсудим в этом разделе, относятся к официальному серверу MySQL от Oracle Corporation, начиная с версии 5.5. У MariaDB MySQL есть несколько связанных оптимизаторов запросов и улучшений подсистемы выполнения, таких как выполнение коррелированных подзапросов изнутри наружу.

Когда стоит использовать коррелированный подзапрос. Не всегда MySQL так уж плохо оптимизирует коррелированные подзапросы. Не слушайте тех, кто советует вам любой ценой избегать их! Вместо этого измеряйте производительность и принимайте решение самостоятельно. Иногда коррелированный подзапрос — вполне приемлемый и даже оптимальный способ получения результата. Рассмотрим пример:

```
mysql> EXPLAIN SELECT film_id, language_id FROM sakila.film
    -> WHERE NOT EXISTS(
    ->    SELECT * FROM sakila.film_actor
    ->    WHERE film_actor.film_id = film.film_id
    -> )\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 951
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: film_actor
         type: ref possible_keys: idx_fk_film_id
          key: idx_fk_film_id
```

```
      key_len: 2
          ref: film.film_id
         rows: 2
        Extra: Using where; Using index
```

Обычно советуют переписать такой запрос с помощью `LEFT OUTER JOIN` вместо подзапроса. Теоретически в обоих случаях план выполнения должен быть одинаков. Посмотрим, так ли это:

```
mysql> EXPLAIN SELECT film.film_id, film.language_id
    -> FROM sakila.film
    ->    LEFT OUTER JOIN sakila.film_actor USING(film_id)
    -> WHERE film_actor.film_id IS NULL\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 951
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref possible_keys: idx_fk_film_id
          key: idx_fk_film_id
      key_len: 2
          ref: sakila.film.film_id
         rows: 2
        Extra: Using where; Using index; Not exists
```

Планы почти идентичны, но все же есть несколько различий.

- Тип `SELECT` для таблицы `film_actor` равен `DEPENDENT SUBQUERY` в одном запросе и `SIMPLE` — в другом. Это различие просто отражает синтаксис, так как в первом запросе используется подзапрос, а во втором — нет. С точки зрения операций обработчика это не слишком существенно.
- В столбце `Extra` для второго запроса нет фразы `Using where` для таблицы `film`. Но и это неважно, поскольку ключевое слово `USING` во втором запросе реализует ту же семантику, что и раздел `WHERE`.
- В столбце `Extra` для второго запроса стоит `Not exists` («Не существует») для таблицы `film_actor`. Это пример работы алгоритма раннего завершения, который мы упоминали ранее в этой главе. Он означает, что MySQL применила оптимизацию типа «не существует» для того, чтобы избежать чтения более чем одной строки из индекса `idx_fk_film_id` для таблицы `film_actor`. Это эквивалентно коррелированному подзапросу `NOT EXISTS()`, поскольку обработка текущей строки прекращается, как только найдено соответствие.

Таким образом, в теории MySQL выполняет запросы почти одинаково. Но на практике только измерение может показать, какой способ быстрее. Мы провели эталонное

тестирование обоих запросов в стандартной конфигурации системы. Результаты приведены в табл. 6.1.

Таблица 6.1. Сравнение NOT EXISTS и LEFT OUTER JOIN

Запрос	Результат, запросов в секунду (QPS)
Подзапрос NOT EXISTS	360
LEFT OUTER JOIN	425

Эталонное тестирование показало, что подзапрос немного медленнее!

Однако так бывает далеко не всегда. Иногда подзапрос может оказаться быстрее. Например, он может оказаться вполне приемлемым, если вам нужно только увидеть строки из одной таблицы, которые соответствуют строкам в другой таблице. Хотя это описание на первый взгляд в точности совпадает с определением соединения, на самом деле это не всегда одно и то же. Следующий запрос с соединением предназначен для поиска фильмов, в которых играет хотя бы один актер, но он возвращает дубликаты, поскольку в некоторых фильмах актеров несколько:

```
mysql> SELECT film.film_id FROM sakila.film
    ->     INNER JOIN sakila.film_actor USING(film_id);
```

Чтобы избавиться от дубликатов, надо использовать ключевое слово `DISTINCT` или раздел `GROUP BY`:

```
mysql> SELECT DISTINCT film.film_id FROM sakila.film
    ->     INNER JOIN sakila.film_actor USING(film_id);
```

Но что же мы на самом деле хотим выразить этим запросом и так ли очевидно наше намерение из SQL-кода? Оператор `EXISTS` выражает логическую идею «имеет соответствие», не создавая строк-дубликатов и не требуя операций `DISTINCT` или `GROUP BY`, для выполнения которых может понадобиться временная таблица. Вот как этот запрос можно переписать с использованием подзапроса вместо соединения:

```
mysql> SELECT film_id FROM sakila.film
    ->     WHERE EXISTS(SELECT * FROM sakila.film_actor
    ->     WHERE film.film_id = film_actor.film_id);
```

Мы провели эталонное тестирование, чтобы определить, какая стратегия быстрее. Результаты представлены в табл. 6.2.

Таблица 6.2. Сравнение EXISTS и INNER JOIN

Запрос	Результат, QPS
INNER JOIN	185
Подзапрос EXISTS	325

Здесь подзапрос выполняется намного быстрее, чем соединение.

Мы привели этот длинный пример, чтобы проиллюстрировать две вещи: во-первых, не стоит слепо доверять категорическим заявлениям о неприемлемости подзапро-

сов, а во-вторых, нужно выполнять измерения, чтобы убедиться в верности своих предположений относительно планов запросов и времени отклика. Последнее замечание о подзапросах: это один из редких случаев, когда нужно упомянуть об ошибке в MySQL. В MySQL 5.1.48 и более ранних версиях следующий синтаксис может блокировать строку в `table2`:

```
SELECT ... FROM table1 WHERE col = (SELECT ... FROM table2 WHERE ...);
```

Эта ошибка, если вы с ней столкнетесь, может привести к тому, что производительность подзапросов будет разной при высоком параллелизме и в случае единственного потока. Это ошибка 46947, и, хотя она уже исправлена, она еще раз подтверждает наш тезис: не предполагайте.

Ограничения UNION

Иногда MySQL не может «опустить» условия с внешнего уровня `UNION` на внутренний, где их можно было бы использовать с целью ограничения результата или создания возможностей для дополнительной оптимизации.

Если вы полагаете, что какой-то из запросов, составляющих `UNION`, будет выполняться быстрее при наличии `LIMIT`, или знаете, что после объединения с другими запросами результаты будут подвергнуты сортировке, то имеет смысл поместить соответствующие ключевые слова в каждую часть `UNION`. Например, в ситуации, когда вы объединяете две очень большие таблицы и ограничиваете результат первыми 20 строками, MySQL сначала запишет обе таблицы во временную, а из нее выберет всего 20 строк:

```
(SELECT first_name, last_name
 FROM sakila.actor
 ORDER BY last_name)
UNION ALL
(SELECT first_name, last_name
 FROM sakila.customer
 ORDER BY last_name)
LIMIT 20;
```

Этот запрос сохранит 200 строк из таблицы актеров и 599 из таблицы клиентов во временной таблице, а затем извлечет из нее первые 20 строк. Этого можно избежать, добавив `LIMIT 20` к каждому запросу внутри `UNION`:

```
(SELECT first_name, last_name
 FROM sakila.actor
 ORDER BY last_name
 LIMIT 20)
UNION ALL
(SELECT first_name, last_name
 FROM sakila.customer
 ORDER BY last_name
 LIMIT 20)
LIMIT 20;
```

Теперь во временной таблице будут храниться только 40 строк. Помимо улучшения производительности, вам, вероятно, придется исправить запрос: порядок, в котором строки извлекаются из временной таблицы, не определен, поэтому перед последним ключевым словом `LIMIT` должен быть общий `ORDER BY`.

Оптимизация слияния индексов

Как сказано в предыдущей главе, MySQL 5.0 и более поздние версии позволяют использовать при выполнении запроса несколько индексов по одной таблице и получать объединение или пересечение результатов для выделения строк, соответствующих всем условиям в разделе `WHERE`.

Распространение равенства

Иногда распространение равенства может вызвать неожиданные затраты. Рассмотрим, к примеру, гигантский список `IN()` для какого-то столбца. Оптимизатору известно, что он равен другим столбцам в других таблицах, благодаря наличию ключевых слов `WHERE`, `ON` или `USING`, устанавливающих равенство столбцов.

Оптимизатор прибегнет к разделению списка, скопировав его во все связанные столбцы в соединенных таблицах. Обычно это полезно, поскольку у оптимизатора запросов и подсистемы выполнения появится больше возможностей определить, где именно проверять сравнение со списком `IN()`. Но если список очень велик, то и оптимизация, и выполнение могут оказаться медленными. В момент написания книги нет никакого обходного пути решения этой проблемы, и если вы с ней столкнетесь, вам придется изменить исходный код (это не проблема для большинства разработчиков).

Параллельное выполнение

MySQL не умеет распараллеливать выполнение одного запроса на нескольких процессорах. Эту возможность предлагают многие серверы баз данных, но только не MySQL. Мы упоминаем о ней лишь для того, чтобы вы не тратили время в попытках понять, как же заставить MySQL выполнять запрос параллельно!

Хеш-соединения

В момент написания книги MySQL не умеет выполнять настоящие хеш-соединения, любое соединение производится методом вложенных циклов. Но можно эмулировать хеш-соединения с помощью хеш-индексов. Правда, если вы работаете не с подсистемой хранения Memory, то и сами хеш-индексы придется эмулировать. Как это делается, мы продемонстрировали в подразделе «Хеш-индексы» в главе 5. MariaDB может выполнить настоящее хеш-соединение.

Непоследовательный просмотр индекса

Исторически MySQL никогда не умела выполнять непоследовательный просмотр индекса, то есть просмотр несмежных диапазонов индекса. При просмотре индекса всегда необходимо задать начальную и конечную точки, даже если для запроса нужны всего несколько несмежных строк в середине диапазона. MySQL просматривает весь диапазон строк между двумя заданными точками.

Поясним это на примере. Предположим, имеется таблица с индексом, построенным по столбцам (`a, b`), и мы хотим выполнить такой запрос:

```
mysql> SELECT ... FROM tbl WHERE b BETWEEN 2 AND 3;
```

Поскольку индекс начинается со столбца `a`, а в условии `WHERE` он не фигурирует, то MySQL вынуждена прибегнуть к полному сканированию таблицы и исключить неподходящие строки с помощью условия `WHERE` (рис. 6.5).

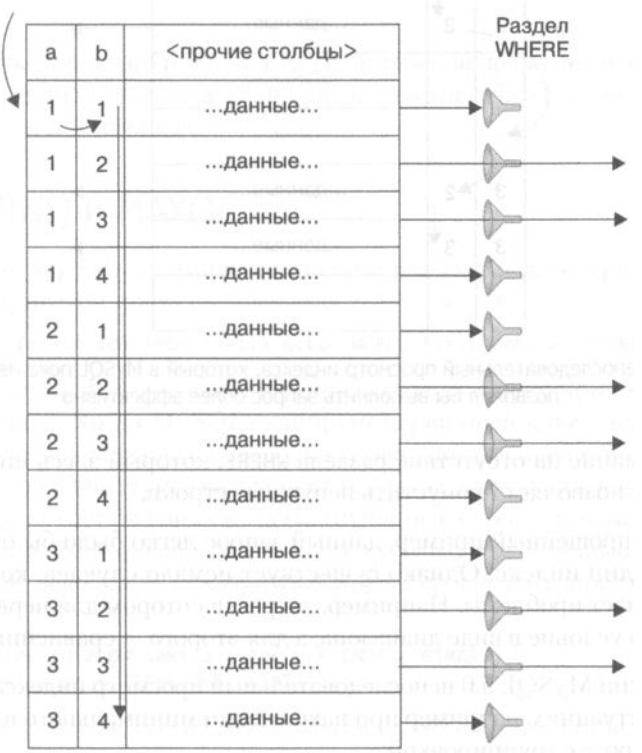


Рис. 6.5. MySQL сканирует всю таблицу в поисках нужных строк

Легко видеть, что существует более быстрый способ выполнения этого запроса. Структура индекса (но не API подсистемы хранения MySQL) позволяет найти начало каждого диапазона значений, просмотреть этот диапазон до конца, затем

вернуться и перейти в начало следующего диапазона. На рис. 6.6 показано, как выглядела бы такая стратегия, если бы MySQL могла претворить ее в жизнь.

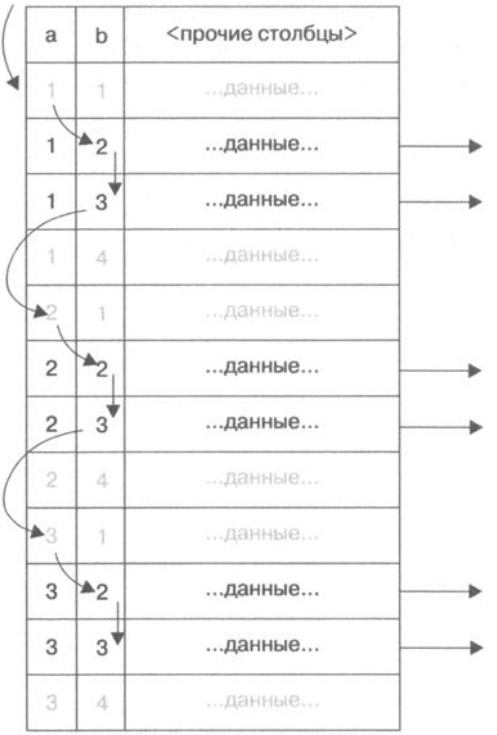


Рис. 6.6. Непоследовательный просмотр индекса, который в MySQL пока не реализован, позволил бы выполнить запрос более эффективно

Обратите внимание на отсутствие раздела `WHERE`, который здесь ни к чему, потому что сам индекс позволяет пропустить ненужные строки.

Конечно, это упрощенный пример; данный запрос легко было бы оптимизировать, добавив еще один индекс. Однако существует немало случаев, когда добавление индекса не решает проблемы. Например, запрос, в котором для первого индексного столбца задано условие в виде диапазона, а для второго — сравнения на равенство.

Начиная с версии MySQL 5.0 непоследовательный просмотр индекса стал возможен в некоторых ситуациях, например при нахождении минимального и максимального значений в запросе с группировкой:

```
mysql> EXPLAIN SELECT actor_id, MAX(film_id)
    -> FROM sakila.film_actor
    -> GROUP BY actor_id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
```

```
          table: film_actor
           type: range possible_keys: NULL
            key: PRIMARY
        key_len: 2
            ref: NULL
           rows: 396
          Extra: Using index for group-by
```

Наличие слов `Using index for group-by` в плане, представленном командой `EXPLAIN`, свидетельствует о непоследовательном просмотре индекса. Для данного частного случая эта оптимизация хороша, но назвать ее универсальным алгоритмом непоследовательного просмотра индекса невозможно. Лучше подошел бы термин «непоследовательное взятие проб из индекса».

До тех пор пока в MySQL не будет реализован универсальный алгоритм, можно применять обходное решение: задавать константу или список констант для столбца, указанного первым в ключе индекса. В предыдущей главе мы привели несколько примеров, показывающих, как этот подход позволяет добиться неплохой производительности.

В MySQL 5.6 некоторые ограничения на сканирование непоследовательного индекса будут исправлены с помощью технологии оптимизатора, называемой *условным индексом с магазинной памятью*.

Функции MIN() и MAX()

MySQL не очень хорошо оптимизирует некоторые запросы, содержащие функции `MIN()` и `MAX()`. Приведем пример:

```
mysql> SELECT MIN(actor_id) FROM sakila.actor WHERE first_name = 'PENELOPE';
```

Поскольку по столбцу `first_name` нет индекса, этот запрос приводит к полному просмотру таблицы. Когда MySQL сканирует первичный ключ, она теоретически может прекратить поиск после нахождения первой подходящей строки, так как значения первичного ключа строго возрастают и, значит, во всех последующих строках значение `actor_id` будет больше уже встретившегося. Однако в этом случае MySQL продолжит сканирование до конца, в чем легко убедиться с помощью счетчиков `SHOW STATUS`. Обходное решение — исключить `MIN()` и переписать запрос с применением `LIMIT`:

```
mysql> SELECT actor_id FROM sakila.actor USE INDEX(PRIMARY)
    -> WHERE first_name = 'PENELOPE' LIMIT 1;
```

Эта общая стратегия зачастую неплохо работает в тех случаях, когда без нее MySQL стала бы просматривать больше строк, чем необходимо. Пуристы могут возразить, что такой запрос входит в противоречие с самой идеей SQL. Предполагается, что мы говорим серверу, что мы хотим получить, а уж он решает, как найти нужные сведения. В данном же случае мы говорим MySQL, как выполнять запрос, при этом из самого запроса неясно, что мы ищем минимальное значение. Это все правда, но иногда для достижения высокой производительности приходится поступаться принципами.

SELECT и UPDATE для одной и той же таблицы

MySQL не позволяет выполнить команду SELECT одновременно с командой UPDATE. На самом деле это не ограничение оптимизатора, но, зная, как MySQL выполняет запросы, вы сможете обойти проблему. Далее приведен пример запроса, написанного в полном соответствии со стандартом SQL, но в MySQL недопустимого. Этот запрос обновляет каждую строку, записывая в нее количество похожих строк, имеющихся в той же таблице:

```
mysql> UPDATE tbl AS outer_tbl
    ->    SET cnt = (
    ->       SELECT count(*) FROM tbl AS inner_tbl
    ->       WHERE inner_tbl.type = outer_tbl.type
    ->    );
ERROR 1093 (HY000): You can't specify target table 'outer_tbl' for update in FROM clause
```

Чтобы обойти это ограничение, можно воспользоваться производной таблицей, поскольку MySQL материализует ее в виде временной таблицы. В результате выполняются два запроса: SELECT в подзапросе и многотабличный UPDATE с соединенными результатами таблицы и подзапроса. Подзапрос открывает и закрывает таблицу перед тем, как внешний UPDATE открывает таблицу, поэтому запрос проходит успешно:

```
mysql> UPDATE tbl
    ->    INNER JOIN(
    ->       SELECT type, count(*) AS cnt
    ->       FROM tbl
    ->       GROUP BY type
    ->    ) AS der USING(type)
    -> SET tbl.cnt = der.cnt;
```

Подсказки оптимизатору запросов

В MySQL имеется множество подсказок оптимизатору запросов, которыми можно воспользоваться, чтобы управлять планом выполнения, если предложенный оптимизатором вас не устраивает. Ниже приведены их перечень и рекомендации, когда имеет смысл применять каждую. Подсказка включается в запрос, чей план выполнения вы хотите изменить, и действует только для этого запроса. Точный синтаксис подсказок можно найти в руководстве пользователя по MySQL. Некоторые из них зависят от номера версии.

- HIGH_PRIORITY и LOW_PRIORITY. Эти подсказки говорят MySQL, какой приоритет назначить данной команде относительно других команд, пытающихся обратиться к тем же таблицам.

  HIGH_PRIORITY означает, что MySQL должна поместить команду SELECT в очередь раньше всех прочих, ожидающих получения блокировок для модификации данных. Фактически команда SELECT должна быть выполнена как можно быстрее, а не ожидать в очереди. Эта подсказка применима и к команде INSERT, в этом случае

она просто отменяет действие глобального параметра LOW_ PRIORITY, установленного на уровне сервера.

Подсказка LOW_PRIORITY оказывает обратное действие: в этом случае команда будет ждать завершения всех остальных команд, желающих обратиться к тем же таблицам, даже если они были отправлены позже. Можно провести аналогию с излишне вежливым человеком: он придерживает дверь в ресторан, пока есть желающие войти, а сам голоден! Вы можете применить данную подсказку к командам SELECT, INSERT, UPDATE, REPLACE и DELETE.

Обе подсказки эффективны в подсистемах хранения, в которых реализована блокировка на уровне таблиц, но в InnoDB и других подсистемах с более детальным контролем блокировки и конкурентного доступа они вам не понадобятся. Будьте осторожны, применяя их к таблицам типа MyISAM, поскольку таким образом можно запретить конкурентные вставки и существенно понизить производительность.

Подсказки HIGH_PRIORITY и LOW_PRIORITY часто понимают неправильно. Их смысл не в том, чтобы выделить запросу больше или меньше ресурсов, чтобы «сервер уделил ему больше внимания», или поменьше, чтобы «сервер не перетруждался». Они просто влияют на дисциплину обслуживания очереди команд, ожидающих доступа к таблице.

- DELAYED. Эта подсказка применяется к командам INSERT и REPLACE. Команда с данной подсказкой возвращает управление немедленно, а подлежащие вставке строки помещаются в буфер и будут реально вставлены все сразу, когда таблица освободится. Это наиболее полезно для журналирования и аналогичных приложений, в которых нужно записывать много строк, не заставляя клиента ждать и не выполняя операцию ввода/вывода для каждой команды в отдельности. Однако у данного режима есть много ограничений. Например, отложенная вставка реализована не во всех подсистемах хранения, а функция LAST_INSERT_ID() в этом случае неприменима.
- STRAIGHT_JOIN. Эта подсказка может встретиться либо сразу после ключевого слова SELECT в команде SELECT, либо в любой другой команде между двумя соединяемыми таблицами. В первом случае она говорит серверу, что указанные в запросе таблицы нужно соединять в порядке перечисления. Во втором — задает порядок соединения таблиц, между которыми находится.

  Подсказка STRAIGHT_JOIN полезна, если выбранный MySQL порядок соединения неоптимален или оптимизатор тратит чересчур много времени на его выбор. В последнем случае поток слишком много времени проводит в состоянии statistics, а включив эту подсказку, можно сократить пространство поиска оптимизатора.

  Использование команды EXPLAIN помогает увидеть, какой порядок выбрал оптимизатор, а затем переписать запрос, расположив таблицы именно в этом порядке, и добавить подсказку STRAIGHT_JOIN. Это хорошая идея, если вы уверены, что фиксированный порядок не ухудшит производительность некоторых условий WHERE. Однако не забывайте пересматривать такие запросы после перехода на новую

версию MySQL, поскольку могут появиться другие оптимизации, подавляемые наличием `STRAIGHT_JOIN`.

- `SQL_SMALL_RESULT` и `SQL_BIG_RESULT`. Эти подсказки применимы только к команде `SELECT`. Они говорят оптимизатору, когда и как использовать временные таблицы или сортировку при выполнении запросов с `GROUP BY` или `DISTINCT`. `SQL_SMALL_RESULT` означает, что результирующий набор будет невелик, так что его можно поместить в индексированную временную таблицу, чтобы не сортировать для группировки. `SQL_BIG_RESULT`, напротив, свидетельствует, что результат велик и лучше использовать временные таблицы на диске с последующей сортировкой.
- `SQL_BUFFER_RESULT`. Эта подсказка говорит оптимизатору, что результаты нужно поместить во временную таблицу и как можно скорее освободить табличные блокировки. Это совсем не то, что буферизация на стороне клиента, описанная нами ранее. Буферизация на стороне сервера может быть полезна, когда вы не используете буферизацию на стороне клиента, поскольку позволяет избежать потребления большого объема памяти клиентом и вместе с тем быстро освободить блокировки. Компромисс состоит в том, что теперь вместо памяти клиента используется память сервера.
- `SQL_CACHE` и `SQL_NO_CACHE`. Эти подсказки говорят серверу о том, является или не является данный запрос кандидатом на помещение в кэш запросов. О том, как ими пользоваться, рассказано в следующей главе.
- `SQL_CALC_FOUND_ROWS`. Эта подсказка, строго говоря, не является подсказкой оптимизатору. Она не говорит MySQL, что нужно иначе планировать запрос. Вместо этого она обеспечивает дополнительную функциональность, изменяя то, что на самом деле делает запрос. Эта подсказка заставляет MySQL вычислить весь результирующий набор, даже если имеется ключевое слово `LIMIT`, ограничивающее количество возвращаемых строк. Получить общее количество строк позволяет функция `FOUND_ROWS()` (однако в подразделе «Оптимизация SQL_CALC_FOUND_ROWS» далее в этой главе говорится о том, почему не следует пользоваться этой подсказкой).
- `FOR UPDATE` и `LOCK IN SHARE MODE`. Эти подсказки также не являются подсказками оптимизатору. Они управляют блокировками для команд `SELECT`, но только в тех подсистемах хранения, где реализованы блокировки на уровне строк. Они позволяют поставить блокировки на найденные строки. Данные подсказки не нужны в запросах вида `INSERT ... SELECT`, поскольку в этом случае MySQL 5.0 и более поздних версий по умолчанию ставит блокировки чтения на строки исходной таблицы. (Это поведение можно отменить, однако так делать не стоит. Мы расскажем о причинах этого в главах, посвященных репликации и резервному копированию.)

  Единственной встроенной подсистемой хранения, поддерживающей эти подсказки, является InnoDB. Применяя их, имейте в виду, что они подавляют некоторые оптимизации, например покрывающие индексы. InnoDB не может монопольно заблокировать строки, не обращаясь к первичному ключу, в котором хранится информация о версиях строк.

К сожалению, этими подсказками часто злоупотребляют, что приводит к серьезным проблемам с блокировкой, что мы обсудим далее в этой главе. Следует избегать их практически любыми средствами — как правило, можно найти лучший способ сделать то, что вы пытаетесь сделать.

- `USE INDEX`, `IGNORE INDEX` и `FORCE INDEX`. Эти подсказки говорят оптимизатору о том, какие индексы использовать или игнорировать при поиске строк в таблице (например, для принятия решения о порядке соединения). В версиях MySQL 5.0 и более ранних версиях они не влияют на выбор сервером индексов для сортировки и группировки. В MySQL 5.1 можно дополнить подсказку ключевыми словами `FOR ORDER BY` или `FOR GROUP BY`.

  `FORCE INDEX` — то же самое, что `USE INDEX`, но эта подсказка сообщает оптимизатору о том, что сканирование таблицы гораздо более затратно, чем поиск по индексу, даже если индекс не очень полезен. Вы можете включить данные подсказки, если полагаете, что оптимизатор выбрал неподходящий индекс, или если хотите по какой-то причине воспользоваться конкретным индексом, например для неявного упорядочения без использования `ORDER BY`. Ранее мы приводили пример, в котором показали, как эффективно получать минимальное значение с помощью ключевого слова `LIMIT`.

В версии MySQL 5.0 и более поздних существуют также несколько конфигурационных переменных, влияющих на поведение оптимизатора.

- `optimizer_search_depth`. Эта переменная указывает оптимизатору, насколько исчерпывающе исследовать частичные планы. Если запрос слишком долго пребывает в состоянии `statistics`, можете попробовать уменьшить это значение.
- `optimizer_prune_level`. Эта переменная, которая по умолчанию включена, позволяет оптимизатору пропускать некоторые планы, основываясь на количестве исследованных строк.
- `optimizer_switch`. Эта переменная содержит набор флагов, которые включают или отключают определенные функции оптимизатора. Например, в MySQL 5.1 вы можете использовать его для отключения плана запросов слияния индекса.

Первые две переменные контролируют режим сокращенного перебора планов выполнения. Такое «срезание углов» бывает полезно для повышения производительности при обработке сложных запросов, но чревато тем, что ради достижения эффективности сервер может пропустить оптимальный план. Поэтому иногда имеет смысл менять их значения.

Проверка обновлений MySQL

Попытка перехитрить оптимизатор MySQL — как правило, неудачная идея. Обычно это приводит к увеличению работы и повышению затрат на обслуживание при очень небольшой выгоде. Это особенно актуально при обновлении MySQL, поскольку

подсказки оптимизатора, применяемые в ваших запросах, могут помешать использованию новых стратегий оптимизации.

В MySQL 5.0 к оптимизатору добавлен ряд возможностей, а еще не выпущенный MySQL 5.6 будет иметь самые большие за продолжительное время изменения в оптимизаторе. Если вы переходите на одну из этих версий, то вряд ли захотите лишиться преимуществ, которые они предлагают.

Новые версии MySQL обычно скачкообразно улучшают работу сервера, и это особенно справедливо для версий 5.5 и 5.6. Обновление MySQL обычно идет хорошо, но вам все равно нужно тщательно протестировать изменения. Всегда есть вероятность того, что вы столкнетесь с экстремальным случаем. Однако мы рады сообщить вам, что такие проблемы можно очень просто предотвратить, приложив небольшие усилия. Используйте инструмент pt-upgrade из пакета Percona Toolkit для проверки того, хорошо ли ваши запросы работают в новой версии MySQL и не возвращают ли они другие результаты.

Оптимизация запросов конкретных типов

В этом разделе мы дадим советы по оптимизации некоторых частных случаев запросов. Большей частью эти темы подробно рассмотрены в других местах книги, но мы хотели представить общий перечень типичных проблем оптимизации, чтобы потом на него можно было ссылаться.

Большинство советов в этом разделе зависят от версии MySQL и к последующим версиям, возможно, будут неприменимы. Нет никаких причин, чтобы в будущем сервер сам не смог производить некоторые или даже все описанные оптимизации.

Оптимизация запросов с COUNT()

Агрегатная функция `COUNT()` и способы оптимизации содержащих ее запросов — это, пожалуй, один из десяти хуже всего понимаемых аспектов MySQL. Поиск в Сети даст столько неправильной информации, что мы даже думать об этом не хотим.

Но прежде, чем переходить к оптимизации, важно понять, что в действительности делает функция `COUNT()`.

Что делает COUNT()

`COUNT()` — это особая функция, которая решает две очень разные задачи: подсчитывает *значения* и *строки*. Значение — это выражение, отличное от `NULL` (`NULL` означает отсутствие какого бы то ни было значения). Если вы укажете имя столбца или какое-нибудь другое выражение в скобках, то `COUNT()` посчитает, сколько раз это

выражение имеет значение. Многие в этом путаются, отчасти потому, что вопрос о значениях и NULL сам по себе не прост. Если вы ощущаете потребность понять, как все это работает, рекомендуем прочитать хорошую книгу об основных SQL. (Интернет вовсе не всегда можно считать надежным источником информации по этой теме.)

Вторая форма COUNT() просто подсчитывает количество строк в результирующем наборе. Так MySQL поступает, когда точно знает, что выражение внутри скобок не может быть равно NULL. Наиболее очевидный пример — выражение COUNT(*), специальная форма COUNT(), которая вовсе не сводится к подстановке вместо специального символа * полного списка столбцов таблицы, как вы, возможно, подумали. На самом деле столбцы вообще игнорируются, а подсчитываются сами строки.

Одна из наиболее часто встречающихся ошибок — задание имени столбца в скобках, когда требуется подсчитать строки. Если вы хотите знать, сколько строк в результирующем наборе, *всегда* используйте COUNT(*). Тем самым вы недвусмысленно сообщите о своем намерении и избежите низкой производительности.

Мифы о MyISAM

Распространено неверное представление о том, что для таблиц типа MyISAM запросы, содержащие функцию COUNT(), выполняются очень быстро. Они быстры, но лишь в очень редком случае при запросе COUNT(*) без условия WHERE, то есть при подсчете общего количества строк в таблице. MySQL может оптимизировать такой запрос, поскольку подсистеме хранения всегда известно, сколько в таблице строк. Если MySQL знает, что столбец col не может содержать NULL, то она может оптимизировать и выражение COUNT(col), самостоятельно преобразовав его в COUNT(*).

MyISAM не обладает никакими магическими возможностями для подсчета строк, когда в запросе есть фраза WHERE, равно как и для более общего случая — подсчета значений, а не строк. Возможно, конкретный запрос она выполнит быстрее, чем другая подсистема хранения, а может, и нет. Это зависит от множества факторов.

Простые оптимизации

Иногда оптимизацию COUNT(*) в MyISAM можно удачно применить, если требуется подсчитать все строки, кроме очень небольшого числа хорошо проиндексированных строк. В следующем примере мы воспользовались стандартной базой данных world, чтобы показать, как можно эффективно подсчитать количество городов с идентификаторами больше 5. Можно было бы записать запрос так:

```
mysql> SELECT COUNT(*) FROM world.City WHERE ID > 5;
```

Если вы проверите его с помощью команды SHOW STATUS, то обнаружите, что он просматривает 4079 строк. Если изменить условие на противоположное и из общего

количества вычесть количество городов с идентификаторами, меньшими либо равными 5, то число просмотренных строк сократится до пяти:

```
mysql> SELECT (SELECT COUNT(*) FROM world.City) - COUNT(*)
    -> FROM world.City WHERE ID <= 5;
```

В этом варианте читается меньше строк, поскольку на стадии оптимизации подзапрос преобразуется в константу, что и показывает `EXPLAIN`:

```
+----+-------------+-------+...+------+-----------------------------+
| id | select_type | table |...| rows | Extra                       |
+----+-------------+-------+...+------+-----------------------------+
|  1 | PRIMARY     | City  |...|    6 | Using where; Using index    |
|  2 | SUBQUERY    | NULL  |...| NULL | Select tables optimized away |
+----+-------------+-------+...+------+-----------------------------+
```

В списках рассылки и IRC-каналах часто спрашивают, как с помощью одного запроса подсчитать, сколько раз встречаются несколько разных значений в одном столбце. Например, вы хотите написать запрос, подсчитывающий предметы разных цветов. Использовать `OR` (например, `SELECT COUNT(color= 'blue' OR color= 'red') FROM items;`) нельзя, потому что невозможно будет разделить счетчики разных цветов. Указать цвета в разделе `WHERE` (например, `SELECT COUNT(*) FROM items WHERE color= 'blue' AND color= 'red';`) тоже нельзя, поскольку цвета являются взаимно исключающими. Задачу тем не менее можно решить с помощью такого запроса[1]:

```
mysql> SELECT SUM(IF(color = 'blue', 1, 0)) AS blue,SUM(IF(color = 'red', 1, 0))
    -> AS red FROM items;
```

А вот еще один эквивалентный запрос, в котором вместо функции `SUM()` используется `COUNT()`. Он основан на том, что выражение не будет иметь значения, когда условие ложно:

```
mysql> SELECT COUNT(color = 'blue' OR NULL) AS blue, COUNT(color = 'red' OR NULL)
    -> AS red FROM items;
```

Использование приближенных значений

Иногда вам не нужен точный подсчет, так что можете использовать приближенные значения. Оценка количества строк, выполненная оптимизатором в команде `EXPLAIN`, как правило, хорошо подходит для этих целей. Просто добавьте к запросу команду `EXPLAIN`.

В других случаях точное количество даже менее эффективно, чем приближенное значение. Один клиент попросил помочь подсчитать количество активных пользователей на своем сайте. Счетчик пользователей кэшировался и отображался в течение 30 минут, после чего обновлялся и снова кэшировался. Тем самым результат, по сути, был неточным. Поэтому приближенное значение было вполне приемлемо. Для того чтобы гарантированно не учитывать неактивных пользователей и пользователя «по умолчанию», у которого в приложении был специальный идентификатор,

[1] Вы также можете написать выражения SUM() в виде SUM(color = 'blue') и SUM(color = 'red').

запрос включал несколько условий WHERE. Удаление этих условий слегка изменило полученную величину, но сделало запрос намного более эффективным. Дальнейшая оптимизация заключалась в устранении ненужного ключевого слова DISTINCT, что позволило избежать файловой сортировки. Переписанный запрос был намного быстрее и возвращал почти те же результаты.

Более сложные оптимизации

В общем случае запросы, содержащие COUNT(), с трудом поддаются оптимизации, поскольку обычно они должны подсчитать много строк (то есть получить доступ к большому объему данных). Единственная возможность оптимизации внутри самой MySQL — использование покрывающих индексов. Если этого недостаточно, стоит внести изменения в архитектуру приложения. Можно рассмотреть вариант создания сводных таблиц (рассмотрены в главе 4) или применить внешнюю систему кэширования, такую как memcached. Возможно, вам придется столкнуться с известной дилеммой: «Быстро, точно, просто — выберите любые два».

Оптимизация запросов с JOIN

Эта тема появляется фактически на протяжении всей книги, но сейчас упомянем несколько важных моментов.

- Убедитесь, что по столбцам, используемым в разделах ON или USING, построены индексы. При добавлении индексов учитывайте порядок соединения. Если таблицы A и B соединяются по столбцу C, а оптимизатор решит, что их надо соединять в порядке B, A, то индексировать таблицу B не обязательно. Неиспользуемые индексы лишь влекут за собой дополнительные расходы. В общем случае следует индексировать только вторую таблицу в порядке соединения, если, конечно, индекс не нужен для каких-то других целей.
- Старайтесь, чтобы в выражениях GROUP BY и ORDER BY встречались столбцы только из одной таблицы, тогда у MySQL появится возможность воспользоваться для этой операции индексом.
- Будьте внимательны при переходе на новую версию MySQL, поскольку в разные моменты изменялись синтаксис соединения, приоритеты операторов и другие аспекты поведения. Конструкция, которая раньше была обычным соединением, вдруг становится декартовым произведением, то есть совершенно другим типом соединения, возвращающим другие результаты, а то и вовсе оказывается синтаксически некорректной.

Оптимизация подзапросов

Самый важный совет, который мы можем дать относительно подзапросов: старайтесь использовать вместо них соединение, по крайней мере в текущих версиях MySQL. Эту тему мы подробно рассматривали в настоящей главе.

Однако совет «предпочитайте соединения» может утратить актуальность, когда выйдут новые версии. Если вы используете MySQL 5.6 или более новых версий либо MariaDB, подзапросы — это совсем другое дело.

Оптимизация GROUP BY и DISTINCT

Во многих случаях MySQL оптимизирует эти два вида запросов схожими способами: фактически она часто конвертирует один в другой на стадии оптимизации. Как всегда, выполнение обоих можно ускорить при наличии подходящих индексов, и это наилучший путь их оптимизации.

Если подходящего индекса не существует, MySQL может применить одну из двух стратегий реализации `GROUP BY`: воспользоваться временной таблицей или прибегнуть к файловой сортировке. Для конкретного запроса более эффективным может оказаться как тот, так и другой подход. Чтобы заставить оптимизатор выбрать нужный вам метод, включите в запрос подсказки `SQL_BIG_RESULT` или `SQL_SMALL_RESULT`, которые были рассмотрены чуть раньше.

Если нужна группировка по значению столбца, который извлекается при соединении из справочной таблицы, то обычно более продуктивно группировать по идентификатору из этой таблицы, а не по его значению. Например, следующий запрос написан не очень удачно:

```
mysql> SELECT actor.first_name, actor.last_name, COUNT(*)
    -> FROM sakila.film_actor
    ->    INNER JOIN sakila.actor USING(actor_id)
    -> GROUP BY actor.first_name, actor.last_name;
```

Лучше написать его так:

```
mysql> SELECT actor.first_name, actor.last_name, COUNT(*)
    -> FROM sakila.film_actor
    ->    INNER JOIN sakila.actor USING(actor_id)
    -> GROUP BY film_actor.actor_id;
```

Группировка по столбцу `actor.actor_id` может оказаться эффективнее, чем по столбцу `film_actor.actor_id`. Чтобы выбрать правильное решение, тестируйте запрос на реальных данных.

В этом примере используется тот факт, что имя и фамилия актера зависят от значения поля `actor_id`, поэтому результаты получатся одинаковыми. Однако не всегда можно так беззаботно включать в список `SELECT` столбцы, по которым не производится группировка, в надежде получить тот же самый эффект. Более того, в конфигурационном файле сервера может быть задан параметр `SQL_MODE`, который вообще запрещает такой режим. Чтобы обойти эту сложность, можно воспользоваться функциями `MIN()` или `MAX()`, если точно известно, что значения в группе различны, так как зависят от группируемого столбца, или если неважно, какое значение будет получено:

```
mysql> SELECT MIN(actor.first_name), MAX(actor.last_name), ...;
```

Пуристы могли бы возразить, что группировка производится не по тому столбцу, что нужно, и были бы правы. Фиктивные `MIN()` или `MAX()` — признак того, что запрос структурирован неправильно. Однако иногда вас волнует только быстрота его выполнения. Пуристы были бы удовлетворены такой формой записи этого запроса:

```
mysql> SELECT actor.first_name, actor.last_name, c.cnt
    -> FROM sakila.actor
    ->    INNER JOIN (
    ->       SELECT actor_id, COUNT(*) AS cnt
    ->       FROM sakila.film_actor
    ->       GROUP BY actor_id
    ->    ) AS c USING(actor_id) ;
```

Однако затраты на создание и заполнение временной таблицы, необходимой для обработки подзапроса, могут оказаться слишком высокими по сравнению с мелким отступлением от принципов реляционной теории. Помните, что у временной таблицы, создаваемой в процессе выполнения подзапроса, нет индексов[1].

В общем случае не стоит включать в список `SELECT` столбцы, по которым не производится группировка, так как результаты получаются недетерминированными и легко могут измениться, если вы создадите другой индекс или оптимизатор выберет иную стратегию. Большинство встречавшихся нам запросов такого рода появились или случайно (поскольку сервер не возражает), или в результате лености программиста, а не потому, что были специально задуманы ради оптимизации. Лучше явно выражать свои намерения. На самом деле мы даже рекомендуем устанавливать конфигурационной переменной `SQL_MODE` режим `ONLY_FULL_GROUP_BY`, чтобы сервер выдавал сообщение об ошибке, а не разрешал писать плохие запросы.

MySQL автоматически упорядочивает результат запроса с группировкой по столбцам, перечисленным в условии `GROUP BY`, если в условии `ORDER BY` явно не указано иное. Если для вас порядок не имеет значения, но вы видите, что в плане выполнения присутствует файловая сортировка, то можете включить в запрос указание `ORDER BY NULL`, которое подавляет автоматическую сортировку. Можно также поместить сразу после `GROUP BY` необязательное ключевое слово `DESC` или `ASC`, задающее направление сортировки.

Оптимизация GROUP BY WITH ROLLUP

Вариацией на тему группирующих запросов является суперагрегирование результатов. Для этого предназначено указание `WITH ROLLUP`, но ее оптимизация может оставлять желать лучшего. С помощью команды `EXPLAIN` проверьте, как выполняется запрос, обращая особое внимание на то, производится ли группировка посредством файловой сортировки или созданием временной таблицы: попробуйте убрать `WITH ROLLUP` и посмотрите, будет ли применен тот же способ группировки. Возможно,

[1] Кстати, это еще одно ограничение, исправленное в MariaDB.

придется принудительно указать метод группировки, включив в запрос вышеупомянутые подсказки.

Иногда оказывается эффективнее выполнить суперагрегирование в самом приложении, даже если для этого придется запросить у сервера больше строк. Можно также воспользоваться вложенным подзапросом в условии `FROM` или создать временную таблицу для хранения промежуточных результатов, а затем выполнить запрос к временной таблице с ключевым словом `UNION`.

Наилучшим подходом станет тот, в котором функциональность `WITH ROLLUP` будет отдана на откуп приложению.

Оптимизация LIMIT и OFFSET

Запросы, содержащие ключевые слова `LIMIT` и `OFFSET`, часто встречаются в системах, выполняющих разбиение на страницы, и почти всегда в сочетании с `ORDER BY`. Полезно иметь индекс, поддерживающий нужное упорядочение, иначе серверу придется слишком часто прибегать к файловой сортировке.

Типичная проблема — слишком большое смещение. Если в запросе встречается фраза `LIMIT 10000, 20`, то сервер сгенерирует 10 020 строк и отбросит первые 10 000, а это очень затратно. В предположении, что доступ ко всем страницам выполняется с одинаковой частотой, такой запрос просматривает в среднем половину таблицы. Для оптимизации можно либо наложить ограничения на то, сколько страниц разрешено просматривать, либо попытаться реализовать обработку больших смещений более эффективно.

Один из простых приемов повышения производительности — выполнять смещение, пользуясь покрывающим индексом, а не исходной таблицей. Затем полученные результаты можно соединить с полными строками, чтобы дополнительно выбрать интересующие столбцы. Такой подход может оказаться намного эффективнее. Рассмотрим следующий запрос:

```
mysql> SELECT film_id, description FROM sakila.film ORDER BY title LIMIT 50, 5;
```

Если таблица очень велика, то его стоит переписать в следующем виде:

```
mysql> SELECT film.film_id, film.description
    -> FROM sakila.film
    ->    INNER JOIN (
    ->       SELECT film_id FROM sakila.film
    ->       ORDER BY title LIMIT 50, 5
    ->    ) AS lim USING(film_id);
```

Это производное соединение работает, поскольку серверу приходится просматривать так мало данных, как это только возможно в индексе, не обращаясь к самим строкам. А после того, как нужные строки найдены, их соединяют с исходной таблицей для того, чтобы сделать выборку недостающих столбцов. Аналогичная методика применима к соединениям с ключевым словом `LIMIT`.

Иногда можно также преобразовать запрос с LIMIT в позиционный запрос, который сервер сможет выполнить путем просмотра диапазона индекса. Например, если предварительно вычислить столбец position и построить по нему индекс, то предыдущий запрос можно будет переписать так:

```
mysql> SELECT film_id, description FROM sakila.film
    -> WHERE position BETWEEN 50 AND 54 ORDER BY position;
```

Похожая проблема возникает при ранжировании данных, причем обычно к этой задаче примешивается раздел GROUP BY. Почти наверняка вам придется заранее вычислять и сохранять ранги.

Проблемой при работе с ключевыми словами LIMIT и OFFSET на самом деле является OFFSET, из-за которого сервер генерирует и отбрасывает строки. Если вы используете какую-то закладку для запоминания позиции последней выбранной строки, то можете сгенерировать следующий набор строк, начиная с этой позиции, вместо использования OFFSET. Например, если вы хотите разбивать на страницы записи об аренде, начиная с последнего заключенного арендного договора и двигаясь назад, то можете быть уверены, что их первичные ключи всегда увеличиваются[1]. Первый набор результатов можно получить следующим образом:

```
mysql> SELECT * FROM sakila.rental
    -> ORDER BY rental_id DESC LIMIT 20;
```

Этот запрос возвращает арендные договоры с идентификаторами с 16049 по 16030. Следующий запрос может продолжить возвращать договоры с этой точки:

```
mysql> SELECT * FROM sakila.rental
    -> WHERE rental_id < 16030
    -> ORDER BY rental_id DESC LIMIT 20;
```

Самое приятное в этой методике то, что она одинаково эффективна независимо от того, в каком месте таблицы вы находитесь.

Альтернативой может стать предварительное вычисление итогов или соединение со вспомогательными таблицами, которые содержат только первичный ключ и столбцы, необходимые для выполнения ORDER BY. Можно также воспользоваться поисковой системой Sphinx (дополнительную информацию см. в приложении Е).

Оптимизация SQL_CALC_FOUND_ROWS

Еще одной широко распространенный методикой оптимизации разбиения на страницы является включение в запрос с ключевым словом LIMIT подсказки SQL_CALC_FOUND_ROWS. Применив этот прием, вы узнаете, сколько строк вернул бы сервер, если бы слова LIMIT не было.

---

[1] Кажется, что происходящее не очень удачно описано. Первичные ключи увеличиваются от начала к концу. Но если двигаться в обратную сторону, то они уменьшаются. — *Примеч. пер.*

Может показаться, будто здесь скрыто какое-то волшебство, позволяющее серверу предсказать, сколько строк он смог бы найти. Увы, на деле все обстоит не так: сервер не умеет подсчитывать строки, которые не отбирал. Это ключевое слово означает лишь, что сервер должен сгенерировать и отбросить оставшуюся часть результирующего набора, а не останавливаться, выбрав затребованное количество строк. Это очень затратно.

Лучше включить в состав элемента разбиения на страницы ссылку на следующую страницу. Предположив, что на странице выводится 20 результатов, мы отбираем с помощью ключевого слова `LIMIT` 21 строку, а выводим только 20. Если 21-я строка существует, значит, имеется следующая страница, и мы формируем ссылку «следующая».

Еще одной возможностью является выбор и кэширование намного большего количества строк, чем необходимо, — скажем, 1000 — и извлечение последующих страниц из кэша. Такая стратегия позволяет приложению узнать размер полного результирующего набора. Если в нем меньше 1000 строк, то точно известно, сколько формировать ссылок на страницы, если больше, то можно просто вывести сообщение: «Найдено более 1000 результатов». Обе стратегии гораздо эффективнее, чем генерация полного набора с последующим отбрасыванием большей его части.

Иногда вы можете просто оценить полный размер результирующего набора, выполнив команду `EXPLAIN` и посмотрев на столбец `rows` в результате (эй, даже Google не показывает точное количество результатов!).

Даже если вы не можете применить ни один из описанных ранее приемов, выполнение отдельного запроса `COUNT(*)` для нахождения количества строк может оказаться намного быстрее режима `SQL_CALC_FOUND_ROWS`, если удастся воспользоваться покрывающим индексом.

Оптимизация UNION

MySQL всегда выполняет запросы с `UNION` путем создания и заполнения временной таблицы. К подобным запросам MySQL может применить не так уж много оптимизаций. Но вы в состоянии вручную помочь оптимизатору, «опустив вниз» фразы `WHERE`, `LIMIT`, `ORDER BY` и другие условия (то есть скопировав их из внешнего запроса в каждый `SELECT`, входящий в `UNION`).

Очень важно *всегда* употреблять `UNION ALL`, если только вы не хотите, чтобы сервер устранял строки-дубликаты. Когда ключевое слово `ALL` отсутствует, MySQL будет создавать временную таблицу в режиме `distinct`, а это значит, что для соблюдения уникальности строки сравниваются целиком. Такая операция весьма затратна. Однако имейте в виду, что наличие слова `ALL` не отменяет необходимости во временной таблице. MySQL обязательно помещает в нее результаты, а затем читает их оттуда, даже если без этого можно обойтись (например, когда результаты можно вернуть напрямую клиенту).

Статический анализ запросов

Пакет Percona Toolkit содержит pt-query-advisor — инструмент, который разбирает журнал запросов, анализирует паттерны запросов и дает раздражающе подробный совет о потенциально опасных приемах в нем. Это своего рода инструмент проверки качества кода для запросов MySQL. Она отловит многие общие проблемы, похожие на те, о которых мы упоминали в предыдущих разделах.

Переменные, определяемые пользователем

О переменных, определяемых пользователем в MySQL, часто забывают, хотя они могут оказаться мощным инструментом при написании эффективных запросов. Особенно хорошо они работают для запросов, в которых можно получить выигрыш от сочетания процедурной и реляционной логики. В чистой реляционной теории все таблицы рассматриваются как неупорядоченные множества, которыми сервер как-то манипулирует целиком. Подход MySQL более прагматичен. Его можно было бы назвать слабой стороной, но если вы знаете, как ею воспользоваться, то сумеете обратить слабость в силу. И тут могут помочь переменные, определяемые пользователем.

Переменные, определяемые пользователем, — это временные контейнеры для значений, чье существование ограничено временем жизни соединения с сервером. Определяются они простым присвоением с помощью команд `SET` или `SELECT`[1]:

```
mysql> SET @one       := 1;
mysql> SET @min_actor := (SELECT MIN(actor_id) FROM sakila.actor);
mysql> SET @last_week := CURRENT_DATE-INTERVAL 1 WEEK;
```

Затем эти переменные можно использовать в различных частях выражений:

```
mysql> SELECT ... WHERE col <= @last_week;
```

Прежде чем начать рассказ о сильных сторонах переменных, определяемых пользователем, рассмотрим некоторые их особенности и недостатки и покажем, для каких целей их нельзя использовать.

- Они подавляют кэширование запроса.
- Их нельзя использовать в тех местах, где требуется литерал или идентификатор, например вместо имени таблицы или столбца либо в разделе `LIMIT`.
- Они связаны с конкретным соединением, поэтому не годятся для передачи информации между соединениями.
- При использовании пула соединений или устойчивых соединений они могут привести к интерференции между, казалось бы, изолированными участками кода. (Это означает, что в коде или в пуле соединений есть ошибка. Тем не менее такое происходит.)

[1] В некоторых контекстах можно использовать просто знак равенства =, но мы считаем, что во избежание неоднозначности лучше всегда употреблять знак :=.

- В версиях, предшествующих MySQL 5.0, они были чувствительны к регистру, поэтому нужно обращать внимание на совместимость.
- Нельзя явно объявить тип переменной, а точки, в которых MySQL принимает решение о типе неопределенной переменной, в разных версиях различаются. Самое правильное — присвоить начальное значение `0` переменным, предназначенным для хранения целых чисел, `0.0` — переменным, предназначенным для хранения чисел с плавающей точкой, и `' '` (пустая строка) — переменным, предназначенным для хранения строк. Тип переменной изменяется в момент присваивания ей значения: в MySQL — динамическая типизация переменных, определяемых пользователем.
- В некоторых ситуациях оптимизатор может вообще исключить такие переменные, поэтому цель их использования не будет достигнута.
- Порядок и даже момент присваивания переменной значения не всегда детерминирован и может зависеть от выбранного оптимизатором плана выполнения. Далее вы увидите, что результат может оказаться совершенно обескураживающим.
- Приоритет оператора присваивания `:=` ниже, чем всех остальных операторов, поэтому следует явно расставлять скобки.
- Наличие неопределенной переменной не приводит к синтаксической ошибке, поэтому легко допустить ошибку, не сознавая этого.

Оптимизация запросов ранжирования

Одной из наиболее важных особенностей переменных является то, что можно одновременно присвоить переменной значение и воспользоваться им. Иными словами, результат операции присваивания — это *L-value*. В следующем примере мы одновременно вычисляем и выводим номер строки:

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum := @rownum + 1 AS rownum
    -> FROM sakila.actor LIMIT 3;
+----------+--------+
| actor_id | rownum |
+----------+--------+
|        1 |      1 |
|        2 |      2 |
|        3 |      3 |
+----------+--------+
```

Сам по себе этот пример не слишком интересен, поскольку просто показывает, как можно продублировать первичный ключ таблицы. Тем не менее и у него есть возможности применения, одна из них — ранжирование строк. Давайте напишем запрос, который возвращает идентификаторы десяти актеров, сыгравших в наибольшем количестве фильмов, причем значение в столбце `rank` должно быть одинаково у актеров, сыгравших в одном и том же количестве фильмов. Начнем с запроса, выбирающего актеров и количество фильмов, в которых они сыграли:

```
mysql> SELECT actor_id, COUNT(*) as cnt
    -> FROM sakila.film_actor
    -> GROUP BY actor_id
```

```
    -> ORDER BY cnt DESC
    -> LIMIT 10;
+----------+-----+
| actor_id | cnt |
+----------+-----+
|      107 |  42 |
|      102 |  41 |
|      198 |  40 |
|      181 |  39 |
|       23 |  37 |
|       81 |  36 |
|      106 |  35 |
|       60 |  35 |
|       13 |  35 |
|      158 |  35 |
+----------+-----+
```

Теперь добавим ранг, который должен быть одинаков у всех актеров, сыгравших в 35 фильмах. Для этого нам понадобятся три переменные: текущий ранг, счетчик фильмов для предыдущего актера и счетчик фильмов для текущего актера. Мы будем изменять ранг при изменении показаний счетчика фильмов. Вот как выглядит первая попытка:

```
mysql> SET @curr_cnt := 0, @prev_cnt := 0, @rank := 0;
mysql> SELECT actor_id,
    ->    @curr_cnt := COUNT(*) AS cnt,
    ->    @rank     := IF(@prev_cnt <> @curr_cnt, @rank + 1, @rank) AS rank,
    ->    @prev_cnt := @curr_cnt AS dummy
    -> FROM sakila.film_actor
    -> GROUP BY actor_id
    -> ORDER BY cnt DESC
    -> LIMIT 10;
+----------+-----+------+-------+
| actor_id | cnt | rank | dummy |
+----------+-----+------+-------+
|      107 |  42 |    0 |     0 |
|      102 |  41 |    0 |     0 |
...
```

Упс! Ранг и счетчик все время остаются равными нулю. Почему?

Невозможно дать универсальный ответ на этот вопрос. Проблема может быть вызвана просто ошибкой при написании имени переменной (в данном случае это не так) или чем-то более сложным. В нашем примере `EXPLAIN` показывает, что применяются временная таблица и файловая сортировка, поэтому переменные вычисляются не в тот момент, когда мы этого ожидаем.

Такое загадочное поведение довольно часто встречается во время работы с пользовательскими переменными в MySQL. Отлаживать подобные ошибки трудно, но затраченные на это усилия окупаются с лихвой. Ранжирование средствами SQL обычно требует квадратичных алгоритмов, таких как подсчет различного количества актеров, сыгравших в большем количестве фильмов, тогда как решение с использованием переменных, определенных пользователем, оказывается линейным, а это существенное улучшение.

Простое решение для данного случая состоит в добавлении в запрос еще одного уровня временных таблиц за счет подзапроса в разделе FROM:

```
mysql> SET @curr_cnt := 0, @prev_cnt := 0, @rank := 0;
    -> SELECT actor_id,
    ->    @curr_cnt := cnt AS cnt,
    ->    @rank     := IF(@prev_cnt <> @curr_cnt, @rank + 1, @rank) AS rank,
    ->    @prev_cnt := @curr_cnt AS dummy
    -> FROM (
    ->    SELECT actor_id, COUNT(*) AS cnt
    ->    FROM sakila.film_actor
    ->    GROUP BY actor_id
    ->    ORDER BY cnt DESC
    ->    LIMIT 10
    -> ) as der;
+----------+-----+------+-------+
| actor_id | cnt | rank | dummy |
+----------+-----+------+-------+
|      107 |  42 |    1 |    42 |
|      102 |  41 |    2 |    41 |
|      198 |  40 |    3 |    40 |
|      181 |  39 |    4 |    39 |
|       23 |  37 |    5 |    37 |
|       81 |  36 |    6 |    36 |
|      106 |  35 |    7 |    35 |
|       60 |  35 |    7 |    35 |
|       13 |  35 |    7 |    35 |
|      158 |  35 |    7 |    35 |
+----------+-----+------+-------+
```

Старайтесь не извлекать только что измененную строку

Что делать, если вы хотите обновить строку, а затем получить некоторую информацию об этом без повторного обращения к ней? К сожалению, MySQL не поддерживает ничего похожего на функционал UPDATE RETURNING, имеющийся в PostgreSQL, который был бы полезен для этой цели. Вместо этого вы можете использовать переменные. Например, один из наших клиентов хотел узнать, как наиболее эффективно присвоить временной метке текущее время, а затем узнать, что это за время. Код выглядел следующим образом:

```
UPDATE t1 SET lastUpdated = NOW() WHERE id = 1;
SELECT lastUpdated FROM t1 WHERE id = 1;
```

Мы использовали переменную и переписали этот запрос следующим образом:

```
UPDATE t1 SET lastUpdated = NOW() WHERE id = 1 AND @now := NOW();
SELECT @now;
```

В обоих случаях применяются два сетевых обращения, но второй запрос не использует таблиц, поэтому он быстрее. (В вашей ситуации скорость может отличаться. Возможно, вам это улучшение не подойдет, но оно делалось для конкретного клиента.)

Подсчет операций UPDATE и INSERT

Что делать, если вы используете `INSERT ON DUPLICATE KEY UPDATE` и хотите узнать, сколько строк было вставлено и не вызвало конфликта с существующими строками, а сколько было обновлений строк? Кристиан Кёнтопп (Kristian Köhntopp) опубликовал решение этой задачи в своем блоге[1]. Суть методики в следующем:

```
INSERT INTO t1(c1, c2) VALUES(4, 4), (2, 1), (3, 1)
ON DUPLICATE KEY UPDATE
    c1 = VALUES(c1) + ( 0 * ( @x := @x +1 ) );
```

Запрос увеличивает значение переменной `@x` при возникновении конфликта, который вызывает выполнение части запроса, содержащей `UPDATE`. Он скрывает значение переменной в выражении, которое умножается на ноль, поэтому переменная не влияет на конечное значение, присвоенное столбцу. Клиентский протокол MySQL возвращает общее количество затронутых строк, поэтому нет необходимости подсчитывать его с помощью пользовательской переменной.

Оценка детерминированного порядка

Большинство проблем в ходе работы с пользовательскими переменными возникает из-за того, что присваивание и чтение значений выполняются на разных стадиях обработки запроса. Например, невозможно предсказать, что произойдет, если присвоить значение в условии `SELECT`, а прочитать — в условии `WHERE`. Возможно, вам покажется, что следующий запрос вернет одну строку, однако в реальности дело обстоит иначе:

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum := @rownum + 1 AS cnt
    -> FROM sakila.actor
    -> WHERE @rownum <= 1;
+----------+------+
| actor_id | cnt  |
+----------+------+
|        1 |    1 |
|        2 |    2 |
+----------+------+
```

Так происходит потому, что разделы `WHERE` и `SELECT` обрабатываются на разных стадиях процесса выполнения запроса. Это будет даже более наглядно, если добавить еще одну стадию, включив раздел `ORDER BY`:

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum := @rownum + 1 AS cnt
    -> FROM sakila.actor
    -> WHERE @rownum <= 1
    -> ORDER BY first_name;
```

[1] См. http://mysqldump.azundris.com/archives/86-Down-the-dirty-road.html.

Этот запрос возвращает все строки из таблицы, поскольку ORDER BY добавил файловую сортировку, а WHERE вычисляется до нее. Решение состоит в том, чтобы присваивать значения и читать их на одной и той же стадии выполнения запроса:

```
mysql> SET @rownum := 0; mysql> SELECT actor_id, @rownum AS rownum
    -> FROM sakila.actor
    -> WHERE (@rownum := @rownum + 1) <= 1;
+----------+--------+
| actor_id | rownum |
+----------+--------+
|        1 | 1      |
+----------+--------+
```

Вопрос на засыпку: что произойдет, если включить в этот запрос фразу ORDER BY? Попробуйте сделать это. Если полученный результат отличается от ожидаемого, то объясните, в чем причина этого. А как насчет следующего запроса, где значение переменной изменяется в разделе ORDER BY, а считывается — в разделе WHERE?

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, first_name, @rownum AS rownum
    -> FROM sakila.actor
    -> WHERE @rownum <= 1
    -> ORDER BY first_name, LEAST(0, @rownum := @rownum + 1);
```

Ответы на большинство вопросов, вызванных неожиданным поведением переменных, определенных пользователем, можно получить, выполнив команду EXPLAIN и поискав фразы `Using where`, `Using temporary` или `Using filesort` в столбце `Extra`.

В последнем примере мы применили еще один полезный трюк: поместили присваивание в функцию LEAST(), так что теперь ее значение отбрасывается и не искажает результатов ORDER BY (мы написали функцию LEAST() таким образом, что она всегда возвращает 0). Этот прием очень полезен, когда присваивание переменной выполняется исключительно ради побочного эффекта, — он позволяет скрыть возвращаемое значение и избежать появления лишних столбцов (таких как dummy в приведенном ранее примере). Для этой цели подходят также функции GREATEST(), LENGTH(), ISNULL(), NULLIF(), COALESCE() и C, используемые поодиночке и в сочетании друг с другом в силу особенностей их поведения. Например, COALESCE() прекращает вычислять свои аргументы, встретив первый имеющий значение, отличное от NULL.

Написание ленивого оператора UNION

Предположим, вы хотите написать запрос UNION, который выполняет первую ветвь UNION и, если находит какие-либо строки, пропускает вторую ветвь. Так можно поступить, если мы ищем строку в таблице, содержащей «горячие» строки (те, к которым обращаются особенно часто), и в таблице с точно такими же строками, к которым, однако, обращаются реже. (Разделение горячих и холодных данных может быть полезным способом повышения эффективности кэширования.)

Приведем пример запроса, который будет искать пользователя в двух местах — в основной таблице пользователя и в таблице пользователей, которые давно не посещали сайт и поэтому для сохранения эффективности были заархивированы[1]:

```
SELECT id FROM users WHERE id = 123
UNION ALL
SELECT id FROM users_archived WHERE id = 123;
```

Это вполне рабочий запрос, однако он будет искать строку в таблице `users_archived`, даже если она найдена в таблице `users`. Мы можем предотвратить это с помощью ленивого оператора `UNION`, который не ленится обратиться ко второй таблице только в том случае, если в первой искомое не обнаружено. При нахождении строки мы задействуем пользовательскую переменную с именем `@found`. Чтобы это произошло, нужно поместить присваивание в список столбцов, поэтому мы используем функцию `GREATEST` в качестве контейнера для присваивания, так что не получим дополнительный столбец в результатах. Чтобы было легче увидеть, из какой таблицы были получены результаты, мы добавим столбец, содержащий имя таблицы. Наконец, нам нужно в конце запроса задать пользовательской переменной значение `NULL`, чтобы у нее не было побочных эффектов и ее можно было использовать повторно. У нас получился такой запрос:

```
SELECT GREATEST(@found := -1, id) AS id, 'users' AS which_tbl
FROM users WHERE id = 1
UNION ALL
   SELECT id, 'users_archived'
   FROM users_archived WHERE id = 1 AND @found IS NULL
UNION ALL
   SELECT 1, 'reset' FROM DUAL WHERE ( @found := NULL ) IS NOT NULL;
```

Прочее использование переменных

Присваивание переменной может встретиться в любой команде, а не только в `SELECT`. Более того, это один из наиболее удачных способов применения переменных, определенных пользователем. Например, вы можете некоторые запросы, такие как вычисление ранга с помощью подзапроса, сделать менее затратными, переписав их в виде команды `UPDATE`.

Впрочем, добиться требуемого поведения не всегда легко. Иногда оптимизатор считает переменные константами этапа компиляции и отказывается выполнять присваивания. Обычно помогает помещение присваиваний внутрь функции типа `LEAST()`. Дадим еще один совет: проверяйте, присвоено ли переменной значение, прежде чем выполнять содержащую ее команду. Иногда у переменной должно быть значение,

[1] Бэрон считает, что некоторые социальные сети архивируют свои данные между его очень редкими посещениями. Когда он входит в систему, его аккаунта, похоже, не существует, но через пару минут приходит электронное письмо с приветствием, и — вуаля! — его аккаунт восстановлен. Это умная оптимизация для асоциальных пользователей, о чем мы поговорим в главе 11.

иногда — нет. Немного поэкспериментировав, вы научитесь делать с помощью переменных, определенных пользователем, очень интересные вещи. Вот несколько идей:

- вычисление промежуточных итогов и средних;
- эмуляция функций `FIRST()` и `LAST()` с помощью запросов с группировкой;
- выполнение математических операций над очень большими числами;
- вычисление MD5-свертки для всей таблицы;
- восстановление выборочного значения, которое оборачивается, если превышает некоторую границу;
- эмуляция курсоров чтения/записи;
- размещение переменных в инструкции `SHOW` путем вставки их в раздел `WHERE`.

Дилемма Си Джей Дейта

Си Джей Дейт (C. J. Date) выступает за подход к созданию базы данных, в рамках которого база данных SQL рассматривается, насколько это возможно, как реляционная база данных. Понятно, как SQL отклоняется от реляционной модели, и, откровенно говоря, MySQL идет дальше, чем некоторые системы управления базами данных. Тем не менее вы не сможете добиться от MySQL хорошей производительности, если попытаетесь заставить ее вести себя как реляционная база данных с помощью некоторых приемов, которые г-н Дейт защищает в своих книгах. К таким приемам относятся, например, глубоко вложенные подзапросы. Это печально, но ограничения MySQL не позволяют более формальному подходу работать хорошо. Мы рекомендуем прочитать его книгу *SQL and Relational Theory: How to Write Accurate SQL Code* (издательство O'Reilly). Это изменит ваше представление о SQL.

Кейс

Иногда речь идет не об оптимизации запросов, схемы, индекса или приложения по отдельности, а о проведении всех этих мероприятий.

Этот кейс посвящен нахождению ответа на вопрос, как решить задачи проектирования, которые часто вызывают вопросы у пользователей. Возможно, вас заинтересует книга Билла Карвина (Bill Karwin) *SQL Antipatterns* (издательство Pragmatic Bookshelf). В ней содержатся рецепты решения конкретных проблем с SQL, которые часто создают трудности для неосторожного программиста.

Построение таблицы очередей в MySQL

Построить таблицу очередей в MySQL непросто, и большинство проектов, которые мы видели, испытывают проблемы с производительностью в условиях большого трафика и при сильном параллелизме. Типичный паттерн для решения этой задачи предполагает создание таблицы, в которой содержится несколько типов строк: необработанные, находящиеся в процессе обработки и обработанные. Один или несколько рабочих процессов ищут необработанные строки, помечают их как «требующие об-

работки», затем выполняют нужные действия и отмечают как обработанные. К типичным объектам с таким поведением можно отнести электронные письма, готовые к отправке, заказы на обработку, комментарии на модерации и т. п.

Можно выделить несколько причин того, почему это плохо работает. Первая: таблица может стать очень большой, и по мере ее роста индексы становятся многоуровневыми, а поиск необработанных строк — медленным. Эту проблему можно решить, разделив очередь на две таблицы, перемещая обработанные строки в архив или таблицу истории и тем самым сохраняя таблицу очередей небольшой.

Вторая причина заключается в том, что поиск строк для обработки, как правило, идет совместно с опросом и блокировкой. Опрос создает нагрузку на сервер, а блокировка вызывает параллелизм и сериализацию между рабочими процессами. В главе 11 мы увидим, почему это усложняет масштабируемость.

Возможно, результат опроса показывает, что все нормально. Но если нет, вы можете оповестить рабочие процессы, что для них есть работа. Например, можно использовать функцию `SLEEP()` с очень длинной задержкой и поясняющим комментарием:

```
SELECT /* ожидание неотправленного письма */ SLEEP(10000);
```

Это вызовет блокировку потока до тех пор, пока не произойдет одно из двух: истекут 10 000 секунд или другой поток подаст команду `KILL QUERY` и завершит ее. Так, после выполнения пакета запросов на вставку в таблицу можно будет посмотреть результаты команды `SHOW PROCESSLIST`, найти потоки, в которых выполняются запросы с вышеуказанным комментарием и принудительно завершить их. Кроме того, вы можете реализовать форму уведомления с функциями `GET_LOCK()` и `RELEASE_LOCK()` или же сделать это за пределами базы данных с помощью службы обмена сообщениями.

Последняя проблема состоит в том, как именно рабочие процессы должны проверять строки, чтобы они не обрабатывались по нескольку раз. Мы неоднократно видели, как это было реализовано с помощью команды `SELECT FOR UPDATE`. Обычно это оказывается очень узким местом масштабируемости и часто вызывает заторы, поскольку транзакции блокируются друг другом и ждут.

В целом стоит избегать команды `SELECT FOR UPDATE`, и не только для таблицы очередей. Эту команду в принципе не стоит использовать. Почти наверняка существует лучший способ достичь желаемой цели. В случае очереди можно использовать простое условие `UPDATE`, чтобы затребовать строки, а затем проверить, затребована ли хотя бы одна из них. Рассмотрим это на примере. Начнем со схемы:

```
CREATE TABLE unsent_emails (
    id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    -- столбцы для сообщений, от кого, кому, тема и т. п.
    status ENUM('unsent', 'claimed', 'sent'),
    owner INT UNSIGNED NOT NULL DEFAULT 0,
    ts    TIMESTAMP,
    KEY   (owner, status, ts)
);
```

Столбец `owner` используется для хранения идентификатора соединения рабочего процесса, который владеет строкой. Это же значение в MySQL возвращает функция `CONNECTION_ID()`. Если это `0`, то строка не востребована.

Мы часто видели такой прием, применяемый, чтобы потребовать десять строк:

```
BEGIN;
SELECT id FROM unsent_emails
    WHERE owner = 0 AND status = 'unsent'
    LIMIT 10 FOR UPDATE;
-- результат: 123, 456, 789
UPDATE unsent_emails
    SET status = 'claimed', owner = CONNECTION_ID()
    WHERE id IN(123, 456, 789);
COMMIT;
```

В данной процедуре используются первые два столбца индекса, поэтому теоретически она выглядит довольно эффективной. Проблема заключается в том, что между двумя запросами приложение имеет некоторое время на обдумывание, из-за чего блокировки строк останавливают работу других клиентов, которые выполняют те же запросы. Все запросы будут использовать один и тот же индекс, поэтому начнут его сканирование с самого начала и, по-видимому, будут мгновенно заблокированы.

Намного эффективнее выполнить запрос следующим образом:

```
SET AUTOCOMMIT = 1;
COMMIT;
UPDATE unsent_emails
    SET status = 'claimed', owner = CONNECTION_ID()
    WHERE owner = 0 AND status = 'unsent'
    LIMIT 10;
SET AUTOCOMMIT = 0;
SELECT id FROM unsent_emails
    WHERE owner = CONNECTION_ID() AND status = 'claimed';
-- результат: 123, 456, 789
```

Вам даже не нужно запускать запрос `SELECT`, чтобы проверить, какие строки вы потребовали. Клиентский протокол расскажет, сколько строк было обновлено, так что вы поймете, были ли затребованы неотправленные строки.

Подобным образом можно переписать большинство запросов `SELECT FOR UPDATE`.

Заключительная часть задачи состоит в очистке строк, которые были затребованы, но не обрабатывались, потому что рабочий процесс по какой-то причине остановился, но ее выполнить легко. Для периодической перезагрузки этих строк можно просто запустить `UPDATE`. Выполните команду `SHOW PROCESSLIST`, составьте список всех идентификаторов потоков, которые в настоящий момент подсоединены к серверу, и укажите их в условии `WHERE`, чтобы избежать потери строки, обрабатываемой прямо сейчас. Предположим, что идентификаторы потоков равны `(10, 20, 30)`. Приведем пример запроса, который «простаивал» и перешел к исправлению строк спустя 10 минут:

```
UPDATE unsent_emails
    SET owner = 0, status = 'unsent'
    WHERE owner NOT IN(0, 10, 20, 30) AND status = 'claimed'
    AND ts < CURRENT_TIMESTAMP - INTERVAL 10 MINUTE;
```

Кстати, обратите внимание на то, как тщательно спроектирован индекс для запускаемых нами запросов. Это пример взаимодействия между материалом данной и предыдущей глав. Показанный запрос может использовать весь индекс, поскольку усло-

вие проверки диапазона помещено в его последний столбец. Данный индекс будет полезен и для других запросов — тем самым мы избежим необходимости применять другой избыточный индекс для двух столбцов, используемых в других запросах.

Этим кейсом проиллюстрированы несколько основополагающих принципов.

- Прекратите это делать или делайте реже. Используйте опрос только в том случае, когда он действительно необходим, поскольку он увеличивает нагрузку и непродуктивную работу.
- Делайте все быстрее. Используйте `UPDATE` вместо `SELECT FOR UPDATE`, за которым следует `UPDATE`, поскольку, чем быстрее выполняется транзакция, тем короче блокировка, меньше параллелизм и сериализация. Кроме того, храните необработанные данные отдельно от обработанных строк.
- Вывод, который можно сделать после рассмотрения этого примера, заключается в том, что отдельные запросы нельзя оптимизировать — их нужно заменить другим запросом или даже другой стратегией. Запросы `SELECT FOR UPDATE` обычно относятся к таким заменяемым запросам.

Иногда лучшим решением является перемещение очереди за пределы сервера базы данных. С очередями хорошо работает Redis, иногда для этой цели можно использовать memcached. В качестве альтернативы можете попробовать подсистему хранения Q4M для MySQL, однако у нас нет опыта ее использования в производственных средах, поэтому мы не можем дать никаких рекомендаций. Кроме того, для некоторых целей могут быть очень полезны RabbitMQ и Gearman[1].

Вычисление расстояния между двумя точками

При работе то и дело приходится сталкиваться с геопространственными вычислениями. Для тяжеловесных вычислений такого рода не принято использовать MySQL — PostgreSQL подойдет лучше. Но все-таки здесь прослеживаются некоторые регулярные закономерности. Например, повсеместно используется запрос, позволяющий найти элементы, расположенные в пределах заданного радиуса от некоторой точки.

Типичные примеры использования такого запроса — поиск квартир в аренду в пределах некоторого расстояния от центра города, фильтрация «совпадений» на сайте знакомств и т. д. Предположим, есть следующая таблица:

```
CREATE TABLE locations (
   id   INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
   name VARCHAR(30),
   lat  FLOAT NOT NULL,
   lon  FLOAT NOT NULL );

INSERT INTO locations(name, lat, lon)
   VALUES('Charlottesville, Virginia', 38.03, -78.48),
         ('Chicago, Illinois',         41.85, -87.65),
         ('Washington, DC',            38.89, -77.04);
```

[1] См. http://www.rabbitmq.com и http://gearman.org/.

Широта и долгота[1] задаются в градусах, и в запросах для нахождения расстояния по поверхности Земли, если считать ее сферой, обычно используется формула большого круга (гаверсинус). Формула для нахождения расстояния между точками *A* и *B* с координатами `latA` и `lonA`, `latB` и `lonB` в радианах может быть выражена следующим образом:

```
ACOS(
    COS(latA) * COS(latB) * COS(lonA - lonB)
      + SIN(latA) * SIN(latB)
)
```

Результат получается в радианах, поэтому, чтобы найти расстояние в милях или километрах, его нужно умножить на радиус Земли, который составляет около 3959 миль, или 6371 км. Предположим, мы хотим найти все точки в радиусе 100 миль от Шарлоттсвилля, где живет Бэрон. Нам нужно преобразовать широту и долготу всех координат в радианы, а затем подставить их в формулу:

```
SELECT * FROM locations WHERE 3979 * ACOS(
  COS(RADIANS(lat)) * COS(RADIANS(38.03)) * COS(RADIANS(lon) - RADIANS(-78.48))
    + SIN(RADIANS(lat)) * SIN(RADIANS(38.03))
) <= 100;
+----+--------------------------+-------+--------+
| id | name                     | lat   | lon    |
+----+--------------------------+-------+--------+
|  1 | Charlottesville, Virginia | 38.03 | -78.48 |
|  3 | Washington, DC           | 38.89 | -77.04 |
+----+--------------------------+-------+--------+
```

Этот тип запроса не только не может использовать индекс, но и будет сжигать тонны циклов процессора и очень сильно загружать сервер. Мы неоднократно сталкивались с этим. Можно ли как-нибудь решить проблему?

В этом проекте есть несколько аспектов, которые можно оптимизировать. Сначала нужно решить, действительно ли необходима такая точность. Существует множество факторов, снижающих точность.

- Объекты могут находиться в зоне радиусом 100 миль, но эта величина может не иметь ни малейшего отношения к фактическому расстоянию до них. Очевидно, независимо от того, где находятся объекты, вы никогда не сможете добраться от одного до другого по абсолютно прямой линии, ведь на пути есть немало препятствий, например большие реки, и чтобы пересечь их, может потребоваться сделать большой крюк в поисках моста. Таким образом, расстояние по карте не слишком точно характеризует фактическое расстояние между двумя объектами.
- Если мы определяем чье-то местоположение на основании его почтового индекса или города, то измеряем расстояние от центра региона до центра другого региона, что также добавляет погрешности вычислениям. Бэрон живет в Шарлоттсвилле, но не в центре города, и ему, вероятно, неинтересно путешествовать точно в центр Вашингтона.

[1] По-английски latitude и longitude соответственно. — *Примеч. пер.*

Возможно, вам действительно нужна точность, но для большинства приложений это лишнее. Это аналогично значимости цифр: вы не можете добиться большей точности результата, чем точность измерительного прибора. (Иначе говоря, «мусор на входе, мусор на выходе».)

Если вам не нужна большая точность, то стоит притвориться, что Земля плоская, а не круглая! Это превращает тригонометрию в гораздо более простое вычисление с применением теоремы Пифагора, которая просто использует пару сумм, произведений и квадратный корень для определения того, находятся ли точки на плоскости на заданном расстоянии друг от друга[1].

Но постойте, разве нам нужен круг? Почему бы просто не использовать квадрат? Углы квадрата с длиной стороны 200 миль находятся всего в 141 миле от его центра, что не сильно отличается от желаемого радиуса 100 миль. Поправим запрос так, чтобы получился квадрат, в котором расстояние от центра до угла будет равно 100 миль (0,0253 радиана):

```
SELECT * FROM locations
WHERE lat BETWEEN  38.03 - DEGREES(0.0253) AND  38.03 + DEGREES(0.0253)
   AND lon BETWEEN -78.48 - DEGREES(0.0253) AND -78.48 + DEGREES(0.0253);
```

Теперь возникает вопрос, как оптимизировать это выражение с помощью индексов. Мы могли бы, конечно, индексировать столбцы `(lat, lon)` или `(lon, lat)`. Но это не очень поможет. Как вы знаете, MySQL 5.5 и более ранних версий не может оптимизировать поиск по столбцам, находящимся правее первого столбца, по которому осуществляется поиск в заданном диапазоне. Другими словами, только один из столбцов будет использоваться эффективно, потому что запрос имеет два условия диапазона (`BETWEEN` эквивалентен значению «больше» и «меньше или равно»).

На помощь снова приходит надежный обходной путь `IN()`. Мы можем добавить два столбца для хранения результата функции `FLOOR()` для каждой координаты, а впоследствии запрос может использовать два списка `IN()` целых чисел для фиксации всех точек, попадающих в нужный квадрат. Вот как можно добавить новые столбцы и индекс, цель которого будет показана в ближайшее время:

```
mysql> ALTER TABLE locations
    ->   ADD lat_floor INT NOT NULL DEFAULT 0,
    ->   ADD lon_floor INT NOT NULL DEFAULT 0,
    ->   ADD KEY(lat_floor, lon_floor);
mysql> UPDATE locations
    -> SET lat_floor = FLOOR(lat), lon_floor = FLOOR(lon);
```

Теперь нужно найти диапазон координат от верхней границы до нижней как на севере, так и на юге. Приведем запрос, который возвращает диапазон градусов, который

[1] Можно еще больше облегчить работу сервера базы данных, если выполнять тригонометрические вычисления в приложении. Тригонометрические функции довольно сильно напрягают процессор. Здорово помочь может, например, перевод всех данных в радианах в приложение и хранение данных в радианах в таблице. Мы старались не слишком усложнять пример и не нагружать его магическими числами неясного происхождения, поэтому не станем показывать эту дополнительную оптимизацию.

мы ищем. Обратите внимание на то, что мы используем запрос только в демонстрационных целях — эти математические вычисления должны выполняться в коде приложения, а не в MySQL:

```
mysql> SELECT FLOOR( 38.03 - DEGREES(0.0253)) AS lat_lb,
    ->      CEILING( 38.03 + DEGREES(0.0253)) AS lat_ub,
    ->        FLOOR(-78.48 - DEGREES(0.0253)) AS lon_lb,
    ->      CEILING(-78.48 + DEGREES(0.0253)) AS lon_ub;
+--------+--------+--------+--------+
| lat_lb | lat_ub | lon_lb | lon_ub |
+--------+--------+--------+--------+
|     36 |     40 |    -80 |    -77 |
+--------+--------+--------+--------+
```

Теперь сгенерируем списки `IN()` со всеми целыми числами между нижней и верхней границами каждого диапазона. Вот этот запрос с дополнительными условиями `WHERE`:

```
SELECT * FROM locations
WHERE lat BETWEEN  38.03 - DEGREES(0.0253) AND  38.03 + DEGREES(0.0253)
   AND lon BETWEEN -78.48 - DEGREES(0.0253) AND -78.48 + DEGREES(0.0253)
   AND lat_floor IN(36,37,38,39,40) AND lon_floor IN(-80,-79,-78,-77);
```

Использование нижней и верхней границ приводит к некоторому усложнению вычислений, поэтому запрос может фактически найти точки, которые лежат вне квадрата. Вот почему нам все еще нужны фильтры на `lat` и `lon`, чтобы отбросить неподходящие результаты. Это похоже на методику моделирования хеш-индекса с помощью столбца `CRC32`, описанную в предыдущей главе: создайте индекс по более или менее достоверному значению, а затем пост-фильтр для удаления нескольких попавших в него неверных значений.

На данном этапе стоит упомянуть, что вместо поиска приближенного квадрата и последующей доработки результатов так, чтобы они соответствовали точному квадрату, мы могли бы находить квадрат, а затем отфильтровывать результаты с помощью формулы большого круга или теоремы Пифагора:

```
SELECT * FROM locations
WHERE lat_floor IN(36,37,38,39,40) AND lon_floor IN(-80,-79,-78,-77)
  AND 3979 * ACOS(
    COS(RADIANS(lat)) * COS(RADIANS(38.03)) * COS(RADIANS(lon) - RADIANS(-78.48))
      + SIN(RADIANS(lat)) * SIN(RADIANS(38.03))
  ) <= 100;
```

Итак, мы вернулись к тому, с чего начали, — получили точный круг, но теперь добились этого эффективнее[1]. Так как вы предварительно фильтруете результирующий набор с помощью эффективных методов, таких как вспомогательные целочисленные столбцы и индексы, вполне допустимо проводить постфильтрацию с применением более затратной математики. Просто не делайте формулу большого круга первым обручем, через который должен перепрыгнуть запрос, иначе все будет работать медленно!

---

[1] Опять же для вычисления выражений, таких как COS (RADIANS (38.03)), следует использовать приложение.

Использовать систему Sphinx, у которой есть несколько хороших встроенных функций геопространственного поиска, по-видимому, намного лучше, чем MySQL. И если вы собираетесь задействовать функции геоинформационных систем (ГИС) MyISAM для методов, показанных в этом разделе, поверьте на слово: они работают ненамного лучше и MyISAM сам по себе неважно работает с большими приложениями со значительным трафиком. Причины этого самые обычные: повреждение данных, блокировка на уровне таблицы и т. д.

В данном кейсе мы рассмотрели обычные стратегии оптимизации.

- Перестаньте выполнять привычные действия или выполняйте реже. Не обсчитывайте весь набор данных с помощью формулы большого круга — сначала сократите его с применением менее затратной методики, а затем уже используйте сложную формулу на меньшем наборе строк.
- Делайте все быстрее. Убедитесь, что спроектированная система позволяет эффективно применять индексы, как описано в предыдущей главе, и для того чтобы избежать ненужной точности, разумно пользуйтесь приближениями (Земля плоская, а квадрат — аппроксимация окружности).
- Перекладывайте максимум работы на плечи приложения. Перенесите все затратные тригонометрические функции из SQL в код приложения!

Применение пользовательских функций

Последняя расширенная оптимизация запросов показывает, когда SQL просто не подходит для выполнения задания. Когда вам нужна просто скорость, ничто не сравнится с кодом на C или C ++. Конечно, вы должны уметь программировать на C или C ++ достаточно хорошо, чтобы не разрушать сервер. Большая сила означает большую ответственность.

Мы покажем, как написать собственные функции, определяемые пользователем (user-defined functions, UDF), в следующей главе, однако думаем, что было бы неплохо рассказать о реальном случае использования UDF уже сейчас. Клиент предъявил к проекту следующее требование: «Нам за несколько секунд нужно выполнить запрос менее чем к 35 миллионам записей, который, по существу, представляет собой операцию XOR между двумя случайными 64-байтовыми длинными строками». Простой расчет показал, что эту задачу просто невозможно решить в MySQL, задействуя имеющееся в настоящий момент оборудование. Как же быть?

Ответом может стать программа, написанная Ивом Трюдо (Yves Trudeau), которая использует набор инструкций SSE4.2. Она работает как служебный процесс на многих промышленных серверах, а сервер MySQL общается с ней по простому сетевому протоколу через UDF, написанному на C. Эталонное тестирование, проведенное Ивом, показало, что сравнение с 4 миллионами строк осуществляется за 130 миллисекунд. Сняв проблему с MySQL и передав ее программе Ива, клиент смог сохранить простоту своего приложения, которое продолжало действовать так, как будто MySQL выполняет всю работу. Как говорится в Twitter, #победа! Это пример оптимизации для бизнеса, а не только решение технических аспектов проблемы.

Итоги главы

Оптимизация запросов — это заключительная деталь направленного на разработку высокопроизводительных приложений пазла, состоящего из схемы, индекса и создания запросов. Чтобы писать хорошие запросы, вам нужно разбираться в схемах и индексировании, и наоборот.

В конечном счете речь все равно идет о времени отклика и необходимости понимать, как выполняются запросы, чтобы иметь возможность рассуждать о том, на что затрачивается время. Добавление пары процессов, таких как синтаксический анализ и оптимизация, — это всего лишь следующий шаг к пониманию того, как MySQL обращается к таблицам и индексам, о которых мы говорили в предыдущей главе. Дополнительный фактор, возникающий, когда вы начинаете изучать взаимодействие между запросами и индексами, — это то, как MySQL обращается к одной таблице или индексу на основе данных, которые она находит в другой таблице.

Оптимизация всегда требует трехстороннего подхода: прекратите это делать, делайте это реже и делайте быстрее. Мы надеемся, что представленные нами кейсы помогут связать все это вместе и продемонстрировать описанный подход в действии.

Помимо фундаментальных строительных блоков, таких как запросы, таблицы и индексы, в MySQL есть дополнительные функциональные возможности, например сегментирование: его цель аналогична цели индексов, но работает оно по-другому. MySQL также поддерживает кэш запросов, что даже позволяет избежать необходимости выполнять запросы (помните, «прекратите это делать»). Мы рассмотрим некоторые из этих функциональных возможностей в следующей главе.

7 Дополнительные возможности MySQL

В версиях MySQL 5.0 и 5.1 появилось немало возможностей, знакомых пользователям других СУБД. Например, секционирование и триггеры. Добавление в MySQL этих возможностей привлекло множество новых пользователей. Однако до начала широкого применения подобных нововведений их влияние на производительность оставалось неясным. В этой главе мы поделимся опытом, полученным в результате использования этих функциональных возможностей на практике, расскажем то, что узнали не из руководств и инструкций.

Секционированные таблицы

Секционированная таблица представляет собой единую логическую таблицу, состоящую из множества физических подтаблиц. Код секционирования — это всего лишь обертка вокруг набора манипуляторов объектов, которая представляет базовые секции и пересылает запросы подсистеме хранения через эти манипуляторы. Секционирование — это своеобразный черный ящик, который скрывает базовые секции на уровне SQL, хотя они легко просматриваются в файловой системе, где вы увидите таблицы компонентов, разделенных в соответствии с соглашением о наименованиях знаком #.

Способ, которым MySQL реализует секционирование как обертку над скрытыми таблицами, означает, что индексы определены для каждой секции, а не для всей таблицы. Этим MySQL отличается, например, от Oracle, в которой индексы и таблицы секционируются более гибкими и сложными способами.

MySQL решает, в какой секции содержатся строки данных, на основе запроса `PARTITION BY`, который вы определяете для таблицы. Оптимизатор запросов может отсечь секции при выполнении запросов, поэтому запросы проверяют не все секции, а лишь те, в которых хранятся искомые данные.

Основная цель секционирования — действовать как грубая форма индексирования и кластеризации данных в таблице. Тем самым можно исключать из рассмотрения большие фрагменты таблицы, а также хранить связанные строки близко друг к другу.

Секционирование обладает множеством достоинств, особенно в следующих ситуациях.

- Когда таблица слишком велика и не помещается полностью в памяти или содержит большое количество хронологической информации, при этом наиболее востребованы строки, находящиеся в ее конце.
- Секционированные данные проще обслуживать, в частности, нетрудно убрать устаревшие значения удалением целой секции. Кроме того, можно оптимизировать проверку и восстановление отдельных секций.
- Секционированные данные можно распределить по физически разным устройствам, так что сервер сможет более эффективно использовать жесткие диски.
- Вы можете использовать секционирование, чтобы избежать некоторых узких мест при определенных рабочих нагрузках, таких как поиндексные мьютексы в InnoDB или блокировки индексных дескрипторов с файловой системой ext3.
- Если это действительно нужно, можете создавать резервные копии и восстанавливать отдельные секции, что может оказаться весьма полезным при чрезвычайно больших наборах данных.

Реализация секционирования в MySQL слишком сложна для того, чтобы описывать ее здесь во всех деталях. Мы хотим обратить внимание лишь на аспекты, относящиеся к производительности, а за базовой информацией отсылаем к руководству по MySQL, в котором очень много материалов по секционированию. Рекомендуем прочитать целиком главу, посвященную этой теме, и заглянуть в разделы, в которых описываются команды `CREATE TABLE`, `SHOW CREATE TABLE`, `ALTER TABLE`, `INFORMATION_SCHEMA.PARTITIONS` `table` и `EXPLAIN`. Из-за секционирования синтаксис команд `CREATE TABLE` и `ALTER TABLE` заметно усложнился.

При использовании секционированных таблиц следует учитывать ряд ограничений. Приведем наиболее важные.

- В таблице не может быть более 1024 секций.
- В MySQL 5.1 выражение секционирования должно быть целым числом или возвращать целое число. В MySQL 5.5 вы можете в определенных случаях выполнить секционирование по столбцам.
- Любой первичный ключ или уникальный индекс должен содержать все столбцы из выражения секционирования.
- Вы не можете использовать ограничения внешнего ключа.

Как работает секционирование

Как уже сказано, в основе секционированных таблиц лежат несколько базовых таблиц, которые представлены манипуляторами объектов. Вы не можете напрямую обращаться к секциям. Каждая секция управляется подсистемой хранения в обычном режиме (все секции должны использовать одну и ту же подсистему хранения), и любые индексы, определенные для этой таблицы, фактически реализуются как идентичные индексы для каждой базовой секции. С точки зрения подсистемы хранения

секции — это просто таблицы, подсистема хранения данных не знает, является ли конкретная таблица, которой она управляет, автономной таблицей или частью более крупной секционированной таблицы.

Действия над секционированной таблицей реализуются с помощью следующих операторов.

- *Запросы SELECT.* Когда вы запрашиваете секционированную таблицу, слой секционирования открывается и блокирует все базовые секции, а оптимизатор запросов определяет, можно ли игнорировать (отсечь) какую-либо из них. Затем с уровня секционирования обработчик вызовов API направляется к подсистеме хранения, которая управляет секционированием.
- *Запросы INSERT.* Когда вы вставляете строку, слой секционирования открывается, блокирует все разделы, определяет, в какую секцию должна быть вставлена строка, и переводит строку туда.
- *Запросы DELETE.* Когда вы удаляете строку, слой секционирования открывается, блокирует все секции, определяет, в какой из них содержится эта строка, и пересылает туда запрос удаления.
- *Запросы UPDATE.* Когда вы изменяете строку, слой секционирования (вы догадались) открывается, блокирует все секции, определяет, в какой из них содержится эта строка, выбирает строку, модифицирует ее, определяет, в какой секции должна содержаться новая строка, пересылает ее с запросом на вставку в целевую секцию и пересылает запрос на удаление в исходную секцию.

Некоторые из этих операций поддерживают обрезку. Например, когда вы удаляете строку, сервер сначала должен ее найти. Если в разделе `WHERE` вы укажете условие, соответствующее выражению секционирования, сервер сможет отсечь секции, в которых не может содержаться искомая строка. То же самое относится к запросам `UPDATE`. Запросы `INSERT` естественным образом отсекаются, сервер просматривает значения, которые нужно вставить, и находит одну и только одну целевую секцию.

Хотя слой секционирования открывается и блокирует все секции, это не означает, что они остаются заблокированными. Подсистема хранения, такая как InnoDB, которая использует собственную блокировку на уровне строк, заставит слой секционирования разблокировать секцию. Этот цикл блокировки и разблокирования аналогичен выполнению запросов к обычным таблицам InnoDB.

Чуть позже мы приведем несколько примеров, которые иллюстрируют затраты и последствия открытия и блокировки каждого раздела в ситуации, когда есть доступ к таблице.

Типы секционирования

MySQL поддерживает несколько типов секционирования. На наш взгляд, наиболее распространенным из них является секционирование диапазонов, в котором каждая секция определена для принятия определенного диапазона значений для некоторого столбца или столбцов или функции, выполненной над значениями этих столбцов.

Приведем пример простого помещения данных по продажам разных лет в отдельные секции:

```
CREATE TABLE sales (
    order_date DATETIME NOT NULL,
    -- Остальные столбцы опущены
) ENGINE=InnoDB PARTITION BY RANGE(YEAR(order_date)) (
     PARTITION p_2010 VALUES LESS THAN (2010),
     PARTITION p_2011 VALUES LESS THAN (2011),
     PARTITION p_2012 VALUES LESS THAN (2012),
     PARTITION p_catchall VALUES LESS THAN MAXVALUE );
```

В разделе секционирования вы можете использовать множество функций. Основное требование заключается в том, что они должны возвращать неконстантное детерминированное целочисленное значение. Здесь использована функция `YEAR()`, но вы можете применять и другие, например `TO_DAYS()`. Секционирование по промежуткам времени — распространенный способ работы с данными на основе даты, поэтому мы вернемся к этому примеру позже и посмотрим, как можно его оптимизировать, чтобы избежать ряда связанных с ним проблем.

Кроме того, MySQL поддерживает методы секционирования по ключам, хеш-секционирование и секционирование списков, часть которых поддерживают подсекции (редко встречаются в реальных условиях). В MySQL вы можете использовать тип секционирования `RANGE COLUMNS`, поэтому вы можете секционировать по столбцам на основе даты, не преобразуя ее в целочисленное значение. Об этом мы поговорим позже.

Однажды мы встречали подсекцию, выделенную для обхода поиндексного мьютекса в InnoDB для таблицы, спроектированной как в нашем последнем примере. Секция для данных последнего года была сильно изменена, что вызвало сильную конкуренцию за этот мьютекс. Выделение подсекций на основе хеша помогло разделить данные на более мелкие части и упростить задачу.

Другие приемы секционирования, которые нам встречались.

- Можно проводить секционирование по ключу для уменьшения конкуренции в мьютексах InnoDB.
- Можно выполнять секционирование по диапазону с помощью функции остатка от деления (по модулю). Так будет создана циклическая таблица, которая сохраняет только желаемый объем данных. Например, при необходимости хранения только данных за последние дни, можно секционировать данные по остатку от деления даты на 7 или просто по дню недели даты.
- Предположим, у вас есть таблица с автоинкрементным столбцом `id`, являющимся ключом, но вы хотите секционировать данные по времени, чтобы горячие последние данные были размещены в одном кластере. Вы не можете секционировать столбец, содержащий временные метки, если не включили его в первичный ключ, но это на корню губит цель первичного ключа. Можете секционировать по выражению, такому как `HASH (id DIV 1000000)`, которое создает новую секцию для каждого миллиона вставленных строк. Тем самым вы

достигаете цели, не меняя первичного ключа. Кроме того, такой подход дает дополнительное преимущество, состоящее в том, что не нужно постоянно создавать секции для хранения новых диапазонов дат, а это пришлось бы делать при секционировании по диапазонам.

Как использовать секционирование

Представьте, что вы хотите запрашивать диапазоны данных из действительно огромной таблицы, которая содержит показатели за многие годы, отсортированные в хронологическом порядке. Вы хотите получить отчеты за последний месяц, что составляет порядка 100 миллионов строк. Через несколько лет эта книга устареет, но давайте притворимся, что вы используете оборудование 2012 года, таблица занимает 10 Тбайт, поэтому она намного больше, чем объем памяти, а у вас есть лишь традиционные жесткие диски, а не флешки (большинство SSD не являются достаточно большими для этой таблицы). Как же вам выполнить запрос к этой таблице вообще, не говоря уж о том, чтобы он был эффективным?

Одно можно сказать наверняка: вы не можете сканировать всю таблицу каждый раз, когда хотите выполнить к ней запрос, поскольку она слишком большая. И вы не хотите использовать индекс из-за того, что его обслуживание влечет за собой затраты и он занимает определенное пространство. В зависимости от индекса, это может привести к сильной фрагментации и плохо кластеризованным данным, а затем и отказу системы вследствие растраты ресурсов на операции хаотичного ввода/вывода. Иногда можно найти обходной путь для одного-двух индексов, но не больше. Тогда остаются только два работоспособных варианта: ваш запрос должен последовательно просканировать часть таблицы либо искомая часть таблицы и индекс должны полностью помещаться в память.

Стоит напомнить, что очень большие индексы, упорядоченные на основе B-дерева, не работают. Если индекс не полностью покрывает запрос, серверу приходится искать полные строки в таблице. Это приводит к произвольному вводу/выводу строки на очень большом пространстве, что просто убьет время отклика на запрос. Затраты на поддержание индекса (дискового пространства, операций ввода/вывода) также очень высоки. Из-за этого такие системы, как Infobright, отбрасывают индексы, упорядоченные на основе B-дерева, и выбирают что-то более крупномодульное, менее затратное для подобных объемов, например блочные метаданные.

Секционирование также может это сделать. Его можно рассматривать как грубую форму индексирования, которая незначительно потребляет ресурсы и позволяет максимально приблизиться к искомым данным. После этого вы можете либо последовательно сканировать соседние строки, либо поместить их в память и проиндексировать. Затраты на секционирование невелики, поскольку отсутствует структура данных, которая указывает на строки и должна периодически обновляться. Секционирование не идентифицирует данные с точностью до строки, и у него нет соответствующей структуры данных. Вместо этого есть условие, в котором указано, какие категории строк в каких секциях могут содержаться.

Рассмотрим две стратегии, которые работают при больших объемах данных.

- *Просматривайте данные, не индексируйте их.* Можете создавать таблицы без индексов и использовать секционирование как единственный механизм для перехода к искомым строкам. Если вы используете раздел `WHERE`, который разделяет запрос на небольшое количество секций, это работает довольно хорошо. Конечно же, вам придется выполнить вычисления и решить, будет ли время отклика на запрос приемлемым. Мы предполагаем, что вы даже не пытаетесь поместить данные в память, кроме того, все, что вы запрашиваете, должно быть прочитано с диска, и эти данные вскоре будут заменены каким-то другим запросом, поэтому кэширование бесполезно. Цель этой стратегии состоит в регулярном получении доступа к большому количеству таблиц. Нужно лишь сделать оговорку: по причинам, которые мы объясним чуть позже, обычно вам придется ограничиваться несколькими сотнями секций.
- *Индексируйте данные и отделяйте часто запрашиваемые.* Если в основном ваши данные используются мало, за исключением некоторой горячей части, то следует секционировать их так, чтобы часто запрашиваемые данные сохранялись в одной секции. И эта секция должна быть довольно мала, чтобы помещаться в памяти вместе со своими индексами, а вы могли бы добавлять индексы и писать запросы, использующие их, как происходит с небольшими таблицами.

Это далеко не все, что вам нужно знать, поскольку у реализации секционирования в MySQL есть несколько ловушек, в которые можно угодить. Давайте посмотрим, что это за ловушки и как их избежать.

Что может пойти не так

Две стратегии секционирования, которые мы только что предложили, основаны на двух ключевых предположениях:

- можно сузить поиск путем отсечения секций при запросе;
- секционирование само по себе не очень затратно.

Как оказалось, эти предположения не всегда справедливы. Вот несколько проблем, с которыми вы можете столкнуться.

- *Значения NULL могут отменить отсечение.* Секционирование работает очень оригинально, когда результатом функции секционирования может быть `NULL`. В этом случае первая секция рассматривается как специальная. Предположим, что вы используете `PARTITION BY RANGE YEAR(order_date)`, как в приведенном ранее примере. Любая строка, у которой в столбце `order_date` окажется значение `NULL` или недействительная дата, будет сохранена в первой определенной вами секции[1]. Теперь предположим, что вы пишете запрос, который заканчивается

[1] Это случится, даже если в столбце order_date не могут храниться значения NULL, поскольку вы можете сохранить значение, не являющееся допустимой датой.

следующим образом: `WHERE order_date BETWEEN '2012-01-01' AND '2012-01-31'`. MySQL фактически проверит не одну секцию а две: секцию, в которой хранятся заказы 2012 года, а также первую секцию в таблице. Она заглянет в первую секцию, поскольку функция `YEAR()` при недопустимых аргументах возвращает `NULL`, следовательно, значения, которые могут соответствовать заданному диапазону, будут храниться как `NULL` в первой секции. То же самое может произойти и с другими функциями, например `TO_DAYS()`[1].

Это может оказаться довольно затратным, если первая секция велика, особенно при использовании стратегии «сканировать, не индексировать». Естественно, проверка двух секций вместо одной нежелательна. Чтобы этого избежать, можно определить фиктивную первую секцию. То есть мы можем исправить предыдущий пример, создав секцию следующим образом: `PARTITION p_nulls VALUES LESS THAN(0)`. Если в таблицу не попадут неверные данные, в этой секции ничего храниться не будет, и хотя ее все-таки станут проверять, это произойдет быстро, потому что она пустая.

Это обходное решение не пригодится в MySQL 5.5, где вы можете секционировать по самому столбцу, а не по связанной с ним функции. В MySQL 5.5 предыдущий пример должен выглядеть так: `PARTITION BY RANGE COLUMNS (order_date)`.

- *Несоответствие PARTITION BY и индекса.* Если вы определяете индекс, который не соответствует разделу секционирования, запросы могут быть неотсекаемыми. Предположим, что вы определяете индекс по столбцу `a` и секцию — по `b`. Каждая секция будет иметь собственный индекс, и поиск по этому индексу откроет и проверит каждое дерево индексов в каждой секции. Этот процесс может быть быстрым, если нелистовые узлы всех индексов находятся в памяти, тем не менее он более затратен чем полное отсутствие индексирования. Чтобы избежать этой проблемы, надо стараться не выполнять индексирование несекционированных столбцов, если ваши запросы также не содержат выражений, которые могут помочь отсечь секции.

  Кажется, что этого очень просто избежать, но действительность может удивить вас. Предположим, что секционированная таблица становится второй таблицей в соединении, а индекс, который применяется для соединения, не используется в разделе запроса, отвечающего за секционирование. В этом случае каждая строка в соединении будет просматривать все секции второй таблицы.

- *Выбор секций может быть затратным.* Различные типы секционирования реализуются по-разному, естественно, их производительность неодинакова. В частности, такие вопросы, как «Где находится эта строка?» или «Где я могу найти строки, соответствующие этому запросу?», могут быть весьма затратными при секционировании по диапазонам, поскольку сервер для нахождения верной секции просматривает список их определений. Как выясняется, этот линейный поиск не так уж эффективен, поскольку затраты растут по мере увеличения количества секций.

[1] Это баг с точки зрения пользователя, но фича с точки зрения разработчика серверной части.

Запросы, которые, по нашему опыту, сильнее всего страдают от этого типа накладных расходов, — это последовательные вставки строк. Для каждой строки, которую вы вставляете в таблицу, секционированную по диапазону, сервер должен проверять список секций, чтобы выбрать место вставки. Эту проблему можно устранить, уменьшив количество секций. Как показывает практика, 100 с небольшим секций довольно хорошо работают в большинстве систем. Другие типы секционирования, такие как секционирование по ключу и хеш-секционирование, не имеют таких ограничений.

- *Открытие и блокировка секций могут быть затратными.* Открытие и блокировка секций во время обращения запроса к секционированной таблице являются другим типом затрат для каждой секции. Открытие и блокировка происходят перед отсечением, поэтому эти затраты не являются отсекаемыми. Этот тип накладных затрат не зависит от типа секционирования и влияет на все типы операторов. Особенно заметен объем накладных расходов для коротких операций, таких как однострочный поиск по первичному ключу. Вы можете избежать высоких затрат на один оператор, выполняя операции блоками, например используя многострочные вставки или `LOAD DATA INFILE`, удаляя диапазоны строк вместо одной за раз и т. д. Кроме того, обязательно, ограничьте количество задаваемых вами секций.
- *Обслуживающие операции могут быть весьма затратными.* Некоторые операции обслуживания секций, например их создание или удаление, очень быстрые. (Отбрасывание базовой таблицы может быть медленным, но это другое дело.) Другие операции, такие как `REORGANIZE PARTITION`, работают аналогично тому, как действует `ALTER`: копируя строки. Например, `REORGANIZE PARTITION` работает, создавая новую временную секцию, перемещая в нее строки и после этого удаляя старую секцию.

Как видите, секционированные таблицы — не панацея. Приведем перечень некоторых ограничений в текущей реализации.

- Всеми секциями должна управлять одна и та же подсистема хранения.
- Существует ряд ограничений на функции и выражения, которые можно использовать в механизме секционирования.
- Некоторые подсистемы хранения вообще не поддерживают секционирование.
- Нельзя пользоваться командой `LOAD INDEX INTO CACHE` в таблицах MyISAM.
- Для таблиц MyISAM для секционированной таблицы требуется больше открытых файловых дескрипторов, чем для обычной таблицы, содержащей такие же данные. Несмотря на то что она выглядит как одна таблица, в действительности это множество таблиц. В результате кэшированная запись одной таблицы может создавать множество дескрипторов файлов. Поэтому, даже если вы настроили кэш таблицы так, что сервер не может превысить максимальное количество файловых дескрипторов операционной системы для каждого процесса, использование секционированных таблиц может привести к тому, что этот предел все равно будет превышен.

Наконец, стоит отметить, что более старые версии серверов хуже новых. Ошибки встречаются во всех программных продуктах. Секционирование появилось в версии MySQL 5.1, и многие ошибки секционирования были зафиксированы уже в версиях 5.1.40 и 5.1.50. В MySQL 5.5 значительно улучшено секционирование для некоторых распространенных на практике случаев. В релизе версии MySQL 5.6 еще больше усовершенствований, например `ALTER TABLE EXCHANGE PARTITION`.

Оптимизация запросов к секционированным таблицам

Секционирование привносит новые способы оптимизации запросов и связанные с ними ловушки. Оптимизатор может использовать функцию секционирования, чтобы отсечь некоторые секции. Как и следовало ожидать от грубого индекса, отсечение позволяет просматривать гораздо меньше данных (в лучшем случае).

Очень важно задавать ключ, по которому производится секционирование, в разделе `WHERE`, даже если он больше ни для чего не нужен, — только при этом условии оптимизатор сможет отсечь ненужные секции. В противном случае подсистема выполнения запроса будет обращаться ко всем секциям в таблице, что при больших таблицах может очень затормозить процесс.

Чтобы понять, отсекает ли оптимизатор секции, можно воспользоваться командой `EXPLAIN PARTITIONS`. Вернемся к уже знакомым тестовым данным:

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales_by_day
   partitions: p_2010,p_2011,p_2012
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3
        Extra:
```

Как видите, этот запрос обращается ко всем секциям. А теперь добавим в раздел `WHERE` условие и посмотрим, что получится:

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day WHERE day > '2011-01-01'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales_by_day
   partitions: p_2011,p_2012
```

Оптимизатор достаточно умен для того, чтобы понять, как производить отсечение, например, он даже может преобразовать диапазон в список дискретных значений и отсечь секции, соответствующие элементам этого списка. Однако он не всеведущ.

Например, следующий раздел WHERE теоретически допускает отсечение, но MySQL этого не сделает:

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day WHERE YEAR(day) = 2010\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales_by_day
   partitions: p_2010,p_2011,p_2012
```

MySQL умеет выполнять отсечение только на основе сравнения со столбцами, указанными в функции секционирования. Она не может принять решение об отсечении по результату вычисления выражения, даже если выражение совпадает с функцией секционирования. Это подобно тому, как индексированные столбцы должны быть изолированы в запросе, чтобы индекс можно было использовать (см. главу 5). Однако предыдущий запрос можно переписать в эквивалентном виде:

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day
    -> WHERE day BETWEEN '2010-01-01' AND '2010-12-31'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales_by_day
   partitions: p_2010
```

Поскольку теперь в условии WHERE явно указан столбец, по которому производилось секционирование, а не выражение, то оптимизатор может спокойно отсечь все секции, кроме одной.

Оптимизатор умеет также отсекать секции во время обработки запроса. Например, если секционированная таблица — вторая в операции соединения, а само соединение производится по ключу секционирования, то MySQL будет искать соединяемые строки только в подходящих секциях. (EXPLAIN не покажет отсечение секции, поскольку это происходит во время выполнения, а не оптимизации запроса.)

Объединенные таблицы

Объединенные таблицы — это своего рода более ранний, более простой вид секционирования с различными ограничениями и меньшим количеством оптимизаций. В то время как секционирование строго фиксирует абстракцию, запрещая разделам доступ к исходным секциям и позволяя ссылаться только на секционированную таблицу, объединенные таблицы позволяют обращаться к базовым таблицам отдельно от них самих. В то время как секционирование больше интегрировано с оптимизатором запросов и является методом будущего, объединенные таблицы почти устарели и однажды даже могут быть удалены.

Подобно секционированным таблицам, объединенные таблицы являются обертками вокруг таблиц MyISAM с такой же структурой. Хотя вы можете считать объединенные таблицы устаревшим и более ограниченным вариантом секционирования, они обладают рядом возможностей, которые у секций отсутствуют[1].

[1] Некоторые люди сравнивают эти возможности с пистолетом, приставленным к виску.

Объединенная таблица — это просто контейнер для реальных таблиц. Имена объединяемых таблиц задаются с помощью ключевого слова UNION в команде CREATE TABLE. Приведем пример, демонстрирующий многие аспекты объединенных таблиц:

```
mysql> CREATE TABLE t1(a INT NOT NULL PRIMARY KEY)ENGINE=MyISAM;
mysql> CREATE TABLE t2(a INT NOT NULL PRIMARY KEY)ENGINE=MyISAM;
mysql> INSERT INTO t1(a) VALUES(1),(2);
mysql> INSERT INTO t2(a) VALUES(1),(2);
mysql> CREATE TABLE mrg(a INT NOT NULL PRIMARY KEY)
    -> ENGINE=MERGE UNION=(t1, t2) INSERT_METHOD=LAST;
mysql> SELECT a FROM mrg;
+------+
| a    |
+------+
|    1 |
|    1 |
|    2 |
|    2 |
+------+
```

Обратите внимание на то, что все объединяемые таблицы должны иметь одинаковое количество и типы столбцов и что все индексы, построенные над объединенной таблицей, существуют и над базовыми таблицами. Это обязательное требование к созданию объединенной таблицы. Отметим также, что в каждой из объединяемых таблиц имеется первичный ключ, состоящий из одного столбца, тем не менее в объединенной таблице есть строки-дубликаты. Это одно из ограничений объединенных таблиц: каждая таблица внутри объединения ведет себя нормально, однако объединенная таблица не проверяет ограничений по всему набору базовых таблиц.

Инструкция INSERT_METHOD=LAST в определении таблицы говорит MySQL о том, что все вставки командой INSERT следует производить в последнюю таблицу. Управлять тем, в какое место объединенной таблицы вставляются новые строки, можно только с помощью этого параметра, который принимает значения LAST или FIRST (хотя вы по-прежнему можете делать вставку непосредственно в объединяемые таблицы). Секционированные таблицы обеспечивают лучшее управление тем местом, где хранятся данные.

Результат выполнения команды INSERT виден и в объединенной таблице, и в одной из базовых:

```
mysql> INSERT INTO mrg(a) VALUES(3);
mysql> SELECT a FROM t2;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
+---+
```

У объединенных таблиц есть ряд других интересных возможностей и ограничений. Например, что происходит, когда сама объединенная таблица или одна из ее базовых удаляется? При удалении объединенной таблицы все ее базовые таблицы остаются

на месте, однако уничтожение любой из базовых таблиц вызывает последствия, зависящие от операционной системы. Например, на платформе GNU/Linux дескриптор файла удаленной таблицы остается открытым и таблица все еще доступна, но только через объединенную таблицу:

```
mysql> DROP TABLE t1, t2;
mysql> SELECT a FROM mrg;
+------+
| a    |
+------+
|    1 |
|    1 |
|    2 |
|    2 |
|    3 |
+------+
```

Существует множество других ограничений и особенностей поведения. Вот некоторые аспекты объединенных таблиц, которые вы должны учитывать.

- Оператор `CREATE`, который создает объединенную таблицу, не проверяет совместимость базовых таблиц. Если базовые таблицы определены несколько иначе, MySQL может создать объединенную таблицу, которую не сможет использовать позже. Кроме того, если вы измените одну из базовых таблиц после создания корректной объединенной таблицы, она перестанет работать, выдав ошибку: `ERROR 1168 (HY000): Unable to open underlying table which is differently defined or of non-MyISAM type or doesn't exist` («Не могу открыть составляющую таблицу, поскольку она определена по-другому, имеет тип, отличный от MyISAM, или не существует»).
- Оператор `REPLACE` для объединенных таблиц не работает вовсе, а `AUTO_INCREMENT` работает не так, как вы ожидаете. Мы предлагаем ознакомиться с деталями, прочитав руководство.
- Запросы, обращенные к объединенной таблице, переадресуются к каждой из базовых таблиц. В результате поиск единственной строки может оказаться более медленным по сравнению с поиском в одной таблице. Поэтому рекомендуется ограничивать количество объединяемых таблиц, особенно если объединенная таблица стоит на втором или последующих местах в операции соединения. Чем меньше количество данных, к которым вы обращаетесь в рамках одной операции, тем выше стоимость доступа к каждой таблице в расчете на операцию в целом. Вот несколько соображений, которые стоит иметь в виду при планировании использования объединенных таблиц:
  - для запросов по диапазону накладные расходы на доступ к базовым таблицам не так существенны, как для запросов на поиск одной строки;
  - сканирование объединенной таблицы выполняется так же быстро, как и обычной;
  - поиск по уникальному и первичному ключу прекращается, как только искомая строка найдена. В данном случае сервер обращается к составляющим таблицам поочередно, пока не найдет нужное значение, после чего оставшиеся таблицы не просматриваются;

- базовые таблицы читаются в порядке, указанном в команде CREATE TABLE. Если вам часто нужно извлекать данные в определенном порядке, этой особенностью можно воспользоваться, чтобы ускорить операцию сортировки слиянием.

Поскольку объединенные таблицы не скрывают базовых MyISAM-таблиц, то существуют некоторые возможности, которых не дает использование секций в MySQL 5.5.

- Одна MyISAM-таблица может входить в несколько объединенных таблиц.
- Базовые таблицы можно переносить с одного сервера на другой, для этого достаточно скопировать файлы с расширениями `.frm`, `.MYI` и `.MYD`.
- В объединенную таблицу легко добавлять базовые таблицы — достаточно создать новую таблицу и изменить определение объединения.
- Можно создать временную объединенную таблицу, которая включает только необходимые данные, например за конкретный период. Секции этого не позволяют.
- Можно исключить таблицу из объединения, если требуется поместить ее в резервную копию, восстановить из копии, изменить определение, исправить или выполнить еще какую-то операцию. Впоследствии таблицу можно вернуть в объединение.
- Команда `myisampack` позволяет сжимать некоторые или все базовые таблицы.

Напротив, части секционированных таблиц скрыты сервером MySQL, доступ к ним можно получить только через саму секционированную таблицу.

Представления

Представления были добавлены в MySQL, начиная с версии 5.0. *Представление* — это виртуальная таблица, в которой не хранятся данные. Вместо этого информация, «находящаяся» в таблице, берется из результатов обработки SQL-запроса, который запускает MySQL, когда вы обращаетесь к представлению. Во многих отношениях MySQL трактует представления точно так же, как таблицы, — они даже разделяют общее пространство имен. Тем не менее в MySQL таблицы и представления — не одно и то же. Например, для представления нельзя создать триггер, и невозможно удалить представление командой DROP TABLE.

В этой книге мы не собираемся объяснять, как создавать и использовать представления, об этом можно прочитать в соответствующем разделе руководства по MySQL. Сосредоточимся на том, как представления реализованы и как взаимодействуют с оптимизатором запросов, чтобы вы поняли, как увеличить их производительность. Для иллюстрации работы представлений воспользуемся демонстрационной базой данных world:

```
mysql> CREATE VIEW Oceania AS
    ->     SELECT * FROM Country WHERE Continent = 'Oceania'
    ->     WITH CHECK OPTION;
```

Чтобы реализовать этот запрос, серверу проще всего выполнить команду SELECT и поместить результаты во временную таблицу. В дальнейшем этой таблицей можно подменять все ссылки на представление. Разберемся, как это могло бы сработать, на примере следующего запроса:

```
mysql> SELECT Code, Name FROM Oceania WHERE Name = 'Australia';
```

Вот как сервер мог бы подойти к его выполнению (имя временной таблицы выбрано исключительно для примера):

```
mysql> CREATE TEMPORARY TABLE TMP_Oceania_123 AS
    ->    SELECT * FROM Country WHERE Continent = 'Oceania';
mysql> SELECT Code, Name FROM TMP_Oceania_123 WHERE Name = 'Australia';
```

Очевидно, при таком подходе не избежать проблем с производительностью и оптимизацией. Более правильный способ реализации представлений — переписать запрос, в котором встречается представление, объединив SQL-код самого запроса с SQL-кодом представления. В следующем примере показано, как мог бы выглядеть исходный запрос после подобного объединения:

```
mysql> SELECT Code, Name FROM Country
    -> WHERE Continent = 'Oceania' AND Name = 'Australia';
```

MySQL может применять оба метода. Для этой цели вызываются два алгоритма: MERGE и TEMPTABLE, причем по возможности MySQL старается использовать алгоритм MERGE. MySQL умеет даже объединять вложенные определения представлений, когда в определении одного представления имеется ссылка на другое. Посмотреть, что получилось в результате переписывания запроса, позволяет команда EXPLAIN EXTENDED, за которой следует команда SHOW WARNINGS.

Если при реализации представления был использован алгоритм TEMPTABLE, то EXPLAIN обычно показывает его как производную (DERIVED) таблицу. На рис. 7.1 показаны обе реализации.

MySQL применяет метод TEMPTABLE, когда в определении представления встречаются ключевые слова GROUP BY, DISTINCT, UNION, агрегатные функции, подзапросы и вообще любые другие конструкции, которые не сохраняют взаимно однозначное соответствие между строками в базовых таблицах и строками в представлении. Приведенный ранее перечень неполон и может измениться в будущем. Чтобы узнать, какой из двух алгоритмов будет применен для конкретного представления, можно выполнить команду EXPLAIN для тривиального запроса SELECT к этому представлению:

```
mysql> EXPLAIN SELECT * FROM <имя_представления>;
+----+-------------+
| id | select_type |
+----+-------------+
|  1 | PRIMARY     |
|  2 | DERIVED     |
+----+-------------+
```

Значение DERIVED в колонке select_type говорит, что для данного представления будет использован метод TEMPTABLE. Однако обратите внимание: если создание базовой производной таблицы весьма затратно, команда EXPLAIN в MySQL 5.5 и более старых версиях также может быть довольно затратной и медленной, поскольку она фактически выполнит и материализует производную таблицу.

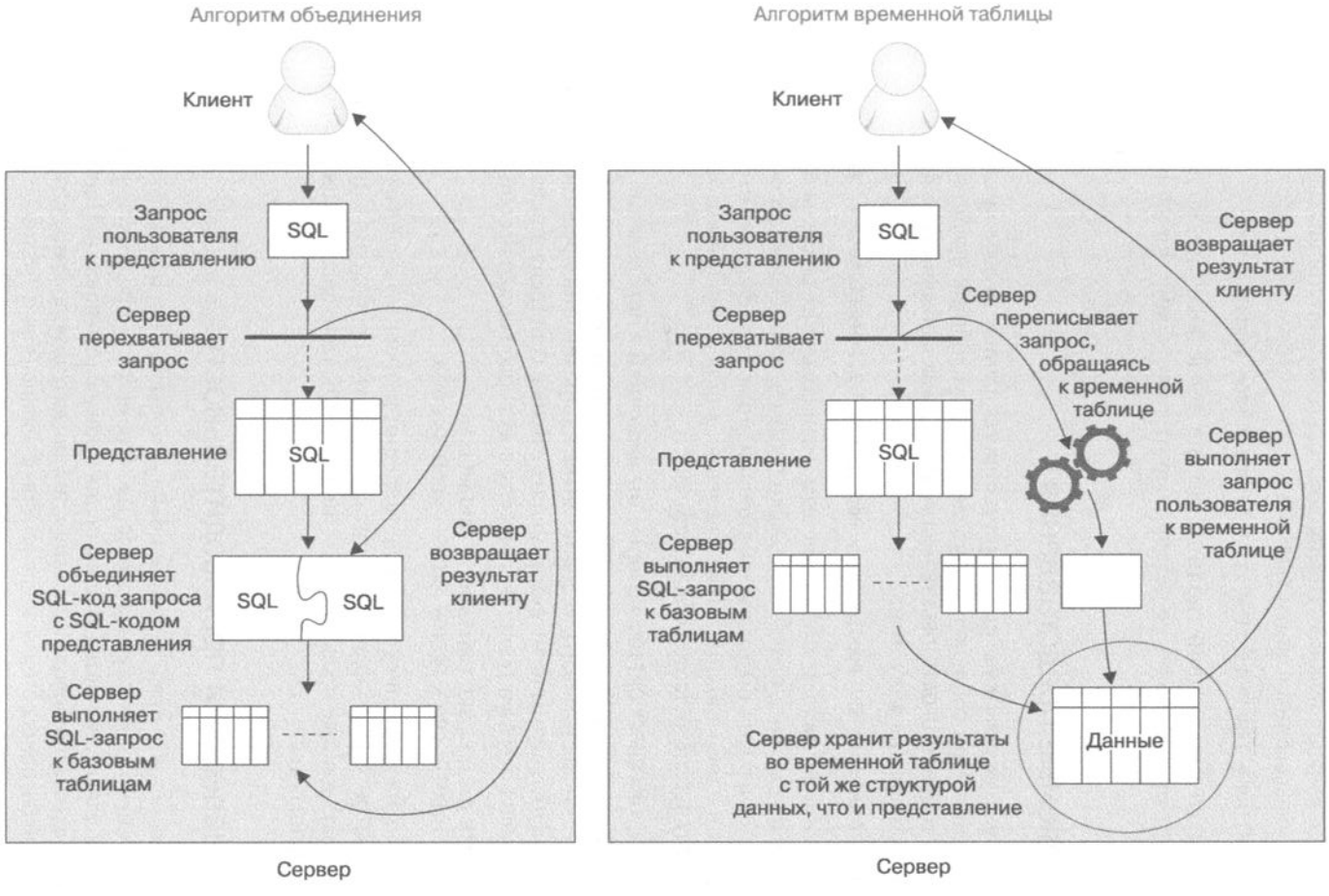


Рис. 7.1. Две реализации представлений

Алгоритм является свойством представления и не зависит от типа запроса, который выполняется к представлению. Предположим, вы создаете тривиальное представление и явно указываете алгоритм `TEMPTABLE`:

```
CREATE ALGORITHM = TEMPTABLE VIEW v1 AS SELECT * FROM sakila.actor;
```

SQL не требует временной таблицы внутри представления, но представление всегда будет использовать ее независимо от того, какой тип запроса к ней выполняется.

Обновляемые представления

Обновляемое представление позволяет изменять данные в базовой таблице через ее представление. При соблюдении определенных условий к представлению можно применять команды `UPDATE`, `DELETE` и даже `INSERT`, как к обычной таблице. Например, следующая операция допустима:

```
mysql> UPDATE Oceania SET Population = Population * 1.1 WHERE Name = 'Australia';
```

Представление не является обновляемым, если содержит разделы `GROUP BY`, `UNION`, агрегатную функцию, а также еще в нескольких случаях. Запрос, изменяющий данные, может содержать операцию соединения, но все изменяемые столбцы должны находиться в одной таблице. Представление, для реализации которого применен метод `TEMPTABLE`, не является обновляемым.

Ключевые слова `CHECK OPTION`, которые мы включили в определение представления из предыдущего раздела, гарантирует, что все строки, измененные через представление, будут соответствовать условию `WHERE` в определении представления и после изменения. Поэтому мы не можем изменять столбец `Continent` или вставлять строку с другим значением `Continent`. В обоих случаях сервер выдал бы ошибку:

```
mysql> UPDATE Oceania SET Continent = 'Atlantis';
ERROR 1369 (HY000): CHECK OPTION failed 'world.Oceania'
```

В некоторых базах данных для представлений разрешено использовать триггеры типа `INSTEAD OF`, что позволяет уточнить, что должно происходить при попытке модифицировать данные представления, но в MySQL триггеры над представлениями не поддерживаются.

Представления и производительность

Многие считают, что представления не могут повысить производительность, однако в MySQL в некоторых случаях это не так. Кроме того, представления могут стать подспорьем для других методов повышения производительности. Например, если рефакторинг схемы происходит поэтапно, то иногда с помощью представлений можно сохранить работоспособность кода, который обращается к таблице с изменившейся структурой.

Вы можете использовать представления для реализации привилегий по столбцу без накладных расходов, связанных с фактическим созданием этих привилегий:

```
CREATE VIEW public.employeeinfo AS
    SELECT firstname, lastname -- но не номер социального страхования
    FROM private.employeeinfo;
GRANT SELECT ON public.* TO public_user;
```

Иногда хороший эффект дают псевдовременные представления. Невозможно создать по-настоящему временное представление, которое существует только в текущем соединении, но можно выбирать для таких представлений специальные имена, быть может, создавать их в особой базе данных, чтобы знать, что впоследствии их можно спокойно удалять. Затем такое представление используется в разделе `FROM` точно так же, как подзапрос. Теоретически оба подхода эквивалентны, но в MySQL для работы с представлениями имеется специальный код, так что временное представление может оказаться более эффективным. Рассмотрим пример:

```
-- Пусть 1234 - результат, возвращенный CONNECTION_ID()
CREATE VIEW temp.cost_per_day_1234 AS
    SELECT DATE(ts) AS day, sum(cost) AS cost
    FROM logs.cost
    GROUP BY day;
SELECT c.day, c.cost, s.sales
FROM temp.cost_per_day_1234 AS c
    INNER JOIN sales.sales_per_day AS s USING(day);
DROP VIEW temp.cost_per_day_1234;
```

Обратите внимание на то, что мы воспользовались идентификатором соединения (`1234`) как уникальным суффиксом, который позволяет избежать конфликта имен. При таком соглашении проще выполнять очистку в случае, когда приложение завершается аварийно и не удаляет временное представление. Дополнительную информацию об этом приеме вы найдете в подразделе «Отсутствующие временные таблицы» на стр. 574.

Производительность представлений, которые реализуются методом `TEMPTABLE`, может оказаться очень низкой, хотя все равно *лучше*, чем у эквивалентного запроса без использования представлений. MySQL выполняет их с помощью рекурсивного шага на стадии оптимизации внешнего запроса, еще до того, как он полностью оптимизирован, поэтому ряд оптимизаций, к которым вы могли привыкнуть, работая с другими СУБД, не применяется. Так, в версии до 5.6 при обработке формирующего временную таблицу запроса условия `WHERE` не «опускается вниз» с уровня внешнего запроса на уровень представления, а индексы над временными таблицами не строятся. Рассмотрим пример, в котором вновь используется представление `temp.cost_per_day_1234`:

```
mysql> SELECT c.day, c.cost, s.sales
    -> FROM temp.cost_per_day_1234 AS c
    ->     INNER JOIN sales.sales_per_day AS s USING(day)
    ->     WHERE day BETWEEN '2007-01-01' AND '2007-01-31';
```

В данном случае сервер выполняет запрос, по которому построено представление, и помещает результат во временную таблицу, а затем соединяет с ней таблицу `sales_per_day`. Встречающееся в условии `WHERE` ограничение `BETWEEN` не включается в представление, поэтому в результирующем наборе представления окажутся все имеющиеся в таблице даты, а не только относящиеся к указанному месяцу. Вдобавок, во временной таблице нет индексов. В данном случае это не составляет проблемы: сервер ставит временную таблицу в соединении первой, поэтому для выполнения соединения можно воспользоваться индексом таблицы `sales_per_day`. Но если бы мы группировали два таких представления, то никаких индексов, позволяющих оптимизировать соединение, не существовало бы.

У представлений есть некоторые проблемы, не относящиеся к MySQL. Они могут обмануть разработчиков, заставляя тех думать, что представления простые, хотя на самом деле их внутреннее устройство очень сложно. Разработчик, который не видит базовой сложности, скорее всего, не подумает о том, что, неоднократно выполняя запрос к тому, что выглядит как таблица, можно нести огромные потери. Нам доводилось видеть, как простой на первый взгляд запрос вызывал появление сотен строк вывода `EXPLAIN`, потому что одна или несколько «таблиц», на которые он ссылался, фактически были представлениями, которые ссылались на множество других таблиц и представлений.

Всякий раз, пытаясь использовать представления для повышения производительности, выполняйте измерения. Даже метод `MERGE` потребляет ресурсы, поэтому заранее трудно сказать, как представление отразится на производительности. Фактически представления используют другой путь выполнения в MySQL-оптимизаторе, который широко не тестируется и может по-прежнему иметь ошибки или проблемы. По этой причине представления не столь идеальны, как хотелось бы. Например, мы видели, как сложные представления при высоком параллелизме заставляли оптимизатор запросов тратить много времени на этапах планирования запроса и сбора статистики, приводя даже к стопору серверов, отладку которых мы выполняли, заменяя представление эквивалентным SQL. Это означает, что представления, даже те, которые используют алгоритм `MERGE`, не всегда реализованы оптимально.

Ограничения представлений

MySQL не поддерживает материализованные представления, с которыми вы, возможно, знакомы по работе с другими СУБД. (В общем случае *материализованное представление* хранит результаты выполнения в невидимой таблице, которую периодически обновляет по исходным данным.) MySQL не поддерживает также индексированные представления. Однако материализованные и индексированные представления можно эмулировать, создав кэшированные или сводные таблицы. Для этих целей вы можете использовать инструмент Flexviews, созданный Джастином Сванхартом (подробнее об этом говорится в главе 4).

Кроме того, в реализации представлений MySQL есть ряд неприятных особенностей. Самая серьезная из них состоит в том, что данная СУБД не сохраняет исходный

SQL-код представления, поэтому, если вы попытаетесь изменить представление, надеясь выполнить команду `SHOW CREATE VIEW`, а затем отредактировать результат, то столкнетесь с неприятным сюрпризом. Запрос будет представлен во внутреннем каноническом виде со всеми кавычками, но без каких бы то ни было форматирования, отступов и комментариев.

Если потребуется отредактировать представление, а исходный красиво отформатированный код утрачен, то его можно найти в последней строке `.frm`-файла, соответствующего данному представлению. Если у вас есть привилегия `FILE`, а `.frm`-файл доступен для чтения всем пользователям, то можно даже загрузить его содержимое с помощью SQL-функции `LOAD_FILE()`. Произведя несложные манипуляции со строками (спасибо Роланду Боумену за изобретательность), можно полностью восстановить исходный код:

```
mysql> SELECT
    ->    REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(
    ->    REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(
    ->       SUBSTRING_INDEX(LOAD_FILE('/var/lib/mysql/world/Oceania.frm'),
    ->       '\nsource=', -1),
    ->    '\\_','\_'), '\\%','\%'), '\\\\','\\'), '\\Z','\Z'), '\\t','\t'),
    ->    '\\r','\r'), '\\n','\n'), '\\b','\b'), '\\\"','\"'), '\\\'','\''),
    ->    '\\0','\0')
    -> AS source;
+-------------------------------------------------------------------------+
| source                                                                  |
+-------------------------------------------------------------------------+
| SELECT * FROM Country WHERE continent = 'Oceania'
   WITH CHECK OPTION
|
+-------------------------------------------------------------------------+
```

Ограничения внешнего ключа

В настоящее время InnoDB — основная подсистема хранения для MySQL, поддерживающая внешние ключи, так что у тех, кому они необходимы, выбор ограничен (они есть еще в PBXT).

Внешние ключи не бесплатное удовольствие. Как правило, их наличие означает, что сервер должен заглядывать в другую таблицу при каждом изменении определенных данных. Хотя для ускорения операции InnoDB принудительно строит индекс, это вовсе не устраняет все негативные последствия подобных проверок. При этом может даже получиться очень большой индекс, имеющий фактически нулевую селективность. Например, предположим, что в огромной таблице имеется столбец `status` и нужно, чтобы он мог содержать только корректные значения, а таковых всего три. Необходимый для этого дополнительный индекс заметно увеличит общий размер таблицы, даже если размер самого столбца мал и в особенности если велика длина первичного ключа. При этом сам индекс нужен лишь для проверки внешнего ключа и больше ни для чего.

И все же в некоторых случаях внешние ключи могут повысить производительность. Если необходимо гарантировать, что данные в двух взаимосвязанных таблицах непротиворечивы, то эффективнее поручить проверку серверу, а не заниматься этим на уровне приложения. Внешние ключи полезны также для каскадного удаления и обновления, хотя эти операции выполняются построчно, то есть медленнее, чем удаление в нескольких таблицах или пакетная операция.

Из-за внешних ключей запрос может распространяться на другие таблицы, а это означает захват блокировок. Например, при попытке вставить строку в дочернюю таблицу ограничение внешнего ключа заставит InnoDB проверить наличие соответствующего значения в родительской таблице. При этом необходимо установить блокировку на строку родительской таблицы, чтобы ее никто не удалил до завершения транзакции. Это может привести к неожиданному ожиданию блокировки и даже к взаимоблокировкам на таблицах, к которым вы напрямую не обращаетесь. Такого рода ошибки далеко не интуитивно понятны, и отлаживать их очень трудно.

Иногда вместо внешних ключей можно использовать триггеры. В таких операциях, как каскадное обновление, внешние ключи работают быстрее триггеров, но если единственное назначение ключа — проверить ограничение, как в примере со столбцом `status`, то, возможно, эффективнее написать триггер, включив в него явный список допустимых значений (можно просто воспользоваться типом данных `ENUM`).

Часто целесообразно проверять ограничения в приложении, а не использовать для этой цели внешние ключи, так как это чревато значительными накладными расходами. У нас нет данных эталонного тестирования, но нам встречалось немало случаев, когда профилирование серверов показало, что проверка ограничений внешнего ключа негативно влияла на производительность, а удаление внешних ключей значительно ее улучшило.

Хранение кода внутри MySQL

MySQL позволяет сохранять код на стороне сервера в форме триггеров, хранимых процедур и хранимых функций. В версии MySQL 5.1 можно делать это также в периодически выполняемых заданиях, которые называются *событиями*. Хранимые процедуры и функции вместе называются *хранимыми подпрограммами*.

Для всех четырех видов хранимого кода применяется специальный расширенный язык SQL, содержащий такие процедурные конструкции, как циклы и условные предложения[1]. Самое существенное различие между разными видами хранимого кода — контекст, в котором этот код выполняется, то есть то, откуда поступают входные данные и куда направляются выходные. Хранимые процедуры и функции могут получать параметры и возвращать результаты, а триггеры и события — нет.

[1] Этот язык является подмножеством языка SQL/PSM (persistent stored modules — «постоянно хранимые модули»), являющегося частью стандарта SQL. Он определен в документе ISO/IEC 9075-4:2003 (E).

В принципе, хранимый код — это неплохой способ совместного и повторного использования кода. Джузеппе Максиа (Giuseppe Maxia) с коллегами написали библиотеку полезных хранимых подпрограмм общего назначения, которая находится по адресу: http://mysql-sr-lib.sourceforge.net. Однако использовать хранимые подпрограммы из других СУБД затруднительно, так как обычно в них применяется собственный язык (исключение составляет DB2, в которой используется очень похожий синтаксис, основанный на том же стандарте)[1].

Мы сфокусируемся на том, как хранимый код отражается на производительности, а не на способах его написания. Если вы планируете писать хранимые процедуры для MySQL, то имеет смысл познакомиться с книгой Гая Харрисона (Guy Harrison) и Стивена Фейерстена (Steven Feuerstein) *MySQL Stored Procedure Programming* (издательство O'Reilly).

Легко найти как сторонников, так и противников применения хранимого кода. Не принимая ничью сторону, мы просто перечислим плюсы и минусы этого подхода в MySQL. Начнем с плюсов.

- Код исполняется там, где находятся данные, поэтому можно сэкономить на сетевом трафике и уменьшить время задержки.
- Это одна из форм повторного использования кода. Она помогает хранить бизнес-правила в одном месте, что обеспечивает согласованное поведение и позволяет избежать лишних треволнений.
- Это упрощает политику выпуска версий и сопровождение.
- Это положительно сказывается на безопасности и позволяет более точно управлять привилегиями. Типичный пример — хранимая процедура для перевода средств с одного банковского счета на другой: она выполняет транзакцию и журналирует ее для последующего аудита. Можно дать приложению право вызывать хранимую процедуру, не открывая доступа к используемым в ней таблицам.
- Сервер кэширует планы выполнения хранимых процедур, что снижает накладные расходы на повторные вызовы.
- Поскольку код содержится на сервере, его можно развертывать, включать в резервную копию и сопровождать средствами сервера. Поэтому хранимый код хорошо приспособлен для задач обслуживания базы данных. У него нет никаких внешних зависимостей, например от библиотек на языке Perl или другого программного обеспечения, которое по тем или иным причинам нежелательно устанавливать на сервере.
- Это способствует разделению труда между программистами приложений и баз данных. Лучше, когда хранимые процедуры пишет специалист по базам данных, поскольку не всякий программист-прикладник умеет создавать эффективные SQL-запросы.

[1] Существуют также утилиты, предназначенные для переноса, например проект tsql2mysql (http://sourceforge.net/projects/tsql2mysql) для переноса из Microsoft SQL Server.

Теперь назовем минусы.

- Вместе с MySQL не поставляются хорошие инструменты разработки и отладки, поэтому писать хранимый код для MySQL труднее, чем для других СУБД.
- Сам язык медленный и примитивный по сравнению с прикладными языками. Количество доступных функций ограниченно, затруднительно программировать сложные манипуляции со строками и нетривиальную логику.
- Наличие хранимого кода может даже усложнить развертывание приложения. Приходится не только вносить изменения в саму программу и схему базы данных, но и развертывать код, хранящийся внутри сервера.
- Поскольку хранимые подпрограммы содержатся в базе данных, могут возникнуть дополнительные уязвимости. Например, реализация нестандартных криптографических функций в хранимой подпрограмме не защитит ваши данные в случае взлома базы. Если бы криптографическая функция была реализована в коде приложения, то хакеру пришлось бы взломать как этот код, так и базу данных.
- Хранимые подпрограммы увеличивают нагрузку на сервер баз данных, а масштабировать его обычно труднее и более затратно, чем приложения или веб-серверы.
- MySQL не предоставляет больших возможностей для управления ресурсами, выделяемыми для исполнения хранимого кода, поэтому ошибка может привести к отказу всего сервера.
- Реализация хранимого кода в MySQL довольно ограниченна — планы выполнения кэшируются на уровне соединения, курсоры материализуются в виде временных таблиц, тяжело было выискивать ошибки в версиях до MySQL 5.5 и т. д. (Мы еще расскажем об ограничениях, присущих отдельным средствам, когда будем их описывать.) В целом язык хранимых подпрограмм хуже, чем T-SQL или PL/SQL.
- Код хранимых процедур в MySQL трудно профилировать. Сложно проанализировать журнал медленных запросов, когда вы видите в нем лишь запись `CALL XYZ('A')`, поскольку придется найти соответствующую процедуру и изучить все команды в ней (это можно настроить в Percona Server).
- Он не очень хорошо работает с двоичным журналированием на основе инструкций или репликацией. В нем так много подводных камней, что, скорее всего, не следует использовать хранимый код с журналированием на основе инструкций, если вы не очень хорошо в нем разбираетесь и не проверяете его на наличие потенциальных проблем.

Мы привели длинный список недостатков, однако что все это значит на практике? Однажды нам встретился случай, когда использование хранимого кода привело к обратным результатам: в одной программе хранимый код использовался с целью создания API для приложения для доступа к базе данных. В результате доступ к базе данных — даже тривиальный поиск строк первичного ключа — осуществлялся через запросы `CALL`, что снизило производительность примерно в пять раз.

В конечном счете хранимый код скрывает сложность. Это упрощает разработку, но зачастую пагубно отражается на производительности и добавляет множество по-

тенциальных проблем с репликацией и другими функциями сервера. Размышляя о том, стоит ли использовать хранимый код, спросите себя: где должна находиться бизнес-логика, в коде приложения или в базе данных? Распространены оба подхода. Нужно лишь понимать, что, прибегая к хранимому коду, вы помещаете логику на сервер базы данных

Хранимые процедуры и функции

Архитектура MySQL и оптимизатора запросов налагает некоторые ограничения на способы использования хранимых подпрограмм и степень их эффективности. В момент работы над этой книгой действовали следующие ограничения.

- Оптимизатор не использует модификатор `DETERMINISTIC` в хранимых функциях для оптимизации нескольких вызовов в одном запросе.
- Оптимизатор не умеет оценивать затраты выполнения хранимой функции.
- Для каждого соединения ведется отдельный кэш планов выполнения хранимых процедур. Если одна и та же процедура вызывается в нескольких соединениях, то ресурсы расходуются впустую на кэширование одного и того же плана. (При использовании пула соединений или постоянных соединений полезная жизнь плана выполнения может быть более длинной.)
- Хранимые процедуры плохо сочетаются с репликацией. Возможно, вы хотите реплицировать не вызов процедуры, а лишь сами изменения в наборе данных. Построчная репликация, появившаяся в версии MySQL 5.1, несколько смягчает эту проблему. Если в MySQL 5.0 включен режим ведения двоичного журнала, то сервер настаивает на том, чтобы вы либо задали для всех хранимых процедур модификатор `DETERMINISTIC`, либо включили параметр с затейливым названием `log_bin_trust_function_creators`.

Мы обычно предпочитаем писать небольшие простые хранимые процедуры. На наш взгляд, сложную логику лучше оставить вне базы данных и реализовывать ее с помощью более выразительного и гибкого процедурного языка. К тому же он может открыть доступ к большему объему вычислительных ресурсов и потенциально позволяет реализовать различные формы кэширования.

Однако для некоторых операций хранимая процедура может оказаться гораздо более быстрой, особенно если речь идет о процедуре, внутри которой есть цикл. В этом случае она может заменить много мелких запросов. Если запрос достаточно мал, то накладные затраты на его разбор и передачу по сети составляют заметную долю всего времени обработки. Чтобы доказать это, мы написали простенькую хранимую процедуру, которая вставляет в таблицу заданное количество строк. Вот ее код:

```
1  DROP PROCEDURE IF EXISTS insert_many_rows;
2
3  delimiter //
4
5  CREATE PROCEDURE insert_many_rows (IN loops INT)
6  BEGIN
```

```
7     DECLARE v1 INT;
8     SET v1=loops;
9     WHILE v1 > 0 DO
10       INSERT INTO test_table values(NULL,0,
11                 'qqqqqqqqqqwwwwwwwwwweeeeeeeeeerrrrrrrrrrttttttttttt',
12                 'qqqqqqqqqqwwwwwwwwwweeeeeeeeeerrrrrrrrrrttttttttttt');
13       SET v1 = v1 - 1;
14     END WHILE;
15 END;
16 //
17
18 delimiter ;
```

Затем мы провели эталонное тестирование и сравнили время вставки 1 миллиона строк с помощью этой процедуры и последовательной вставки из клиентского приложения. Структура таблицы и используемое оборудование значения не имеют, важно лишь относительное быстродействие при разных подходах. Просто ради интереса мы определили, сколько времени уходит на те же запросы при соединении с помощью MySQL Proxy. Чтобы не усложнять ситуацию, эталонные тесты прогоняли на одном сервере, где находились также клиентское приложение и экземпляр MySQL Proxy. Результаты представлены в табл. 7.1.

Таблица 7.1. Общее время вставки 1 миллиона строк по одной

Метод	Общее время, с
Хранимая процедура	101
Клиентское приложение	279
Клиентское приложение с соединением через MySQL Proxy	307

Хранимая процедура работает гораздо быстрее главным образом из-за отсутствия накладных затрат на передачу данных по сети, разбор, оптимизацию и т. д.

Пример типичной хранимой процедуры для задач обслуживания базы данных мы приведем позже в этой главе.

Триггеры

Триггеры дают возможность выполнить код, когда встречаются команды `INSERT`, `UPDATE` или `DELETE`. Вы можете заставить MySQL обрабатывать триггеры до и/или после выполнения самой команды. Триггер не возвращает значения, но в состоянии читать и/или изменять данные, которые были модифицированы командой, вызвавшей его срабатывание. Следовательно, с помощью триггеров можно реализовывать ограничения целостности или бизнес-логику, которую иначе пришлось бы программировать на стороне клиента.

Триггеры позволяют упростить логику приложения и повысить производительность, поскольку избавляют от необходимости обмениваться данными по сети между клиентом и сервером. Они бывают полезными также для автоматического обновления

денормализованных и сводных таблиц. Так, в демонстрационной базе Sakila триггеры применяются для поддержания таблицы `film_text` в актуальном состоянии.

В настоящее время реализация триггеров в MySQL весьма ограниченна. Даже если у вас есть обширный опыт работы с триггерами в других базах данных, не стоит предполагать, что в MySQL они будут работать точно так же. В частности, отметим следующие моменты.

- В каждой таблице для каждого события можно определить только один триггер (иными словами, не может быть двух триггеров, срабатывающих по событию `AFTER INSERT`).
- MySQL поддерживает только триггеры на уровне строки, то есть триггер всегда работает в режиме `FOR EACH ROW`, а не для команды в целом. При обработке больших наборов данных это гораздо менее эффективно.

Следующие недостатки триггеров относятся и к MySQL.

- Триггеры могут затруднить понимание того, что в действительности делает сервер, поскольку простая на первый взгляд команда может инициировать большой объем невидимой работы. Например, если триггер обновляет связанную таблицу, то количество затрагиваемых командой строк может удвоиться.
- Триггеры трудно отлаживать. При их использовании зачастую очень сложно найти узкие места, ухудшающие производительность.
- Триггеры могут стать причиной неочевидных взаимных блокировок и ожиданий блокировок. Если в триггере возникает ошибка, то с ошибкой завершается и исходный запрос, а если вы не знаете о существовании триггера, то разобраться, о чем говорит код ошибки, будет затруднительно.

С точки зрения производительности самое серьезное ограничение — это реализация в MySQL только триггеров `FOR EACH ROW`. Иногда из-за этого триггер работает настолько медленно, что оказывается непригоден для поддержания сводных и кэшированных таблиц. Основной причиной использования триггеров вместо периодического массового обновления состоит в том, что данные в любой момент времени согласованы.

Кроме того, триггеры не всегда гарантируют атомарность. Например, действия, выполненные триггером при обновлении таблицы типа MyISAM, невозможно откатить, если в команде, ставшей причиной его срабатывания, обнаружится ошибка. Да и сам триггер может вызывать ошибки. Предположим, что вы присоединили триггер типа `AFTER UPDATE` к таблице в MyISAM и используете его для обновления другой таблицы MyISAM. Если из-за ошибки в триггере вторую таблицу обновить не удастся, то откат обновления первой не будет выполнен.

Триггеры над таблицами типа InnoDB выполняются в рамках текущей транзакции, поэтому их действия будут атомарными — они фиксируются или откатываются вместе с вызвавшей их командой. Однако если триггер таблицы InnoDB проверяет данные в другой таблице, чтобы проконтролировать ограничение целостности, то следует помнить о механизме MVCC, так как иначе можно получить неправильные результаты. Пусть, например, требуется эмулировать внешние ключи, но вы не хотите

использовать механизм внешних ключей, встроенный в InnoDB. Можно написать триггер типа `BEFORE INSERT`, который проверит наличие соответствующей записи в другой таблице, но если читать в триггере данные из нее не с помощью команды `SELECT FOR UPDATE`, то в случае конкурентного обновления этой таблицы можно получить неверные результаты.

Мы вовсе не хотим отговаривать вас от использования триггеров. Напротив, они бывают очень полезны, особенно для проверки ограничений, задач обслуживания системы и поддержания денормализованных данных в актуальном состоянии.

Триггеры можно применять и с целью журналирования обновлений строк. Это полезно для написанных пользователем схем репликации, когда требуется разорвать соединение между двумя системами, изменить данные, а затем восстановить синхронизацию. Простой пример — группа пользователей, которые берут с собой ноутбуки в другой офис. Сделанные ими изменения нужно затем синхронизировать с главной базой данных, после чего скопировать главную базу обратно на отдельные ноутбуки. Эта задача требует двусторонней синхронизации. Триггеры неплохо подходят для построения подобных систем. На каждом ноутбуке с помощью триггеров все операции модификации данных журналируются в специальные таблицы с указанием того, какие строки изменились. Затем специально написанная программа синхронизации переносит изменения в главную базу данных. А уже потом с помощью стандартной репликации MySQL можно синхронизировать ноутбуки с главной базой, в результате чего на каждом переносном компьютере окажутся изменения, выполненные на всех остальных ноутбуках. Однако вам нужно быть очень осторожными с триггерами, вставляющими строки в другие таблицы, которые автоматически увеличивают первичные ключи. Это не очень хорошо работает с репликацией на основе команд, так как значения автоинкремента, вероятно, будут отличаться друг от друга в разных подчиненных серверах.

Иногда удается даже обойти ограничение `FOR EACH ROW`. Роланд Боумен (Roland Bouman) обнаружил, что функция `ROW_COUNT()` внутри триггера возвращает `1` везде, кроме первой строки в триггере типа `BEFORE`. Этим фактом можно воспользоваться для того, чтобы подавить выполнение триггера для каждой измененной строки, активизировав его лишь один раз на всю команду. Это, конечно, не триггер уровня команды, но все-таки полезный прием, позволяющий в некоторых случаях эмулировать триггер `BEFORE` уровня команды. Возможно, описанная особенность на самом деле является ошибкой, которая рано или поздно будет исправлена, поэтому относитесь к этому приему с осторожностью и проверяйте, работает ли он после перехода на новую версию. Приведем пример того, как можно использовать эту недокументированную возможность:

```
CREATE TRIGGER fake_statement_trigger
BEFORE INSERT ON sometable
FOR EACH ROW
BEGIN
    DECLARE v_row_count INT DEFAULT ROW_COUNT( );
    IF v_row_count <> 1 THEN
        -- Здесь должен быть ваш код
    END IF;
END;
```

События

События — это новый вид хранимого кода, появившийся в MySQL, начиная с версии 5.1. Они похожи на задания cron, но выполняются целиком внутри сервера MySQL. Можно создать событие, которое будет выполнять SQL-код в нужный момент времени или с заданным интервалом. Обычно поступают так: оформляют сложный SQL-код в виде хранимой процедуры, а событие просто вызывает ее с помощью команды `CALL`.

События инициируются специальным потоком планировщика событий, поскольку никак не связаны с соединениями. Они не принимают параметров и не возвращают значений просто потому, что их неоткуда получить и некому вернуть. Выполненные событием команды фиксируются в журнале сервера, если он активизирован, но понять, что они были вызваны именно из события, затруднительно. Можно также заглянуть в таблицу `INFORMATION_SCHEMA.EVENTS` и посмотреть на состояние события, например узнать, когда оно в последний раз вызывалось.

К событиям применимы те же соображения, что и к хранимым процедурам, — это дополнительная нагрузка на сервер. Накладные расходы на сам запуск события минимальны, но SQL-код, который запускается событием, может заметно повлиять на производительность. Кроме того, события могут вызывать те же самые проблемы с репликацией на основе команд, что и другой хранимый код. Разумно использовать события для периодического запуска задач обслуживания, в том числе перестроения кэшированных и сводных таблиц, эмулирующих материализованные представления, а также для сохранения переменных состояния с целью мониторинга и диагностики.

В следующем примере создается событие, которое один раз в неделю запускает хранимую процедуру в указанной базе данных (как создать эту хранимую процедуру, покажем чуть позже):

```
CREATE EVENT optimize_somedb ON SCHEDULE EVERY 1 WEEK
DO
CALL optimize_tables('somedb');
```

Можно указать, следует ли реплицировать события на подчиненный сервер. Иногда это необходимо, иногда — нет. Возьмем только что рассмотренный пример: возможно, вы хотите выполнять операцию `OPTIMIZE TABLE` на всех подчиненных серверах, но имейте в виду, что производительность сервера в целом может пострадать (в частности, из-за установки табличных блокировок), если все подчиненные серверы начнут выполнять эту операцию в одно и то же время.

Наконец, если для завершения выполнения периодического события требуется длительное время, то может случиться, что оно в очередной раз активизируется в момент, когда предыдущее еще не завершилось. MySQL не защищает вас от этого, так что придется написать собственную логику взаимного исключения. Чтобы гарантировать, что работает только один экземпляр события, можно воспользоваться функцией `GET_LOCK()`:

```
CREATE EVENT optimize_somedb ON SCHEDULE EVERY 1 WEEK
DO
BEGIN
    DECLARE CONTINUE HANLDER FOR SQLEXCEPTION
        BEGIN END;
```

```
    IF GET_LOCK('somedb', 0) THEN
        DO CALL optimize_tables('somedb');
    END IF;
    DO RELEASE_LOCK('somedb');
END
```

Пустой обработчик продолжения гарантирует, что событие освободит блокировку, даже если хранимая процедура выбросит исключение.

Хотя события никак не связаны с соединениями, ассоциация между ними и потоками все же имеется. Существует главный поток планировщика событий, который необходимо активизировать в конфигурационном файле сервера или командой `SET`:

```
mysql> SET GLOBAL event_scheduler := 1;
```

Будучи активизирован, этот поток выполнит события согласно расписанию, определенному в этом событии. Вы можете просмотреть журнал ошибок сервера для получения информации о выполнении события.

Хотя планировщик событий выполняется в одном потоке, события могут запускаться конкурентно. Сервер создает новый процесс для каждого события. Внутри кода события функция `CONNECTION_ID()`, как обычно, возвращает уникальное значение, хотя соединения как такового не существует (функция `CONNECTION_ID()` на самом деле возвращает просто идентификатор потока). Процесс и поток будут жить только во время выполнения события. Вы можете увидеть его в выводе команды `SHOW PROCESSLIST`, посмотрев на столбец `Command`, который появится как Connect.

Хотя процесс обязательно создает поток для фактического выполнения, последний уничтожается в конце выполнения события и не помещается в кэш потока, а счетчик состояния `Threads_created` не увеличивается.

Сохранение комментариев в хранимом коде

Код хранимых процедур и функций, триггеров и событий может быть довольно длинным, поэтому неплохо снабдить его комментариями. Но комментарии не могут храниться на сервере, поскольку стандартный клиент командной строки может вырезать их (эта особенность командного клиента, возможно, вызывает у вас раздражение, но се ля ви).

Есть полезный прием, который поможет оставить комментарии в хранимом коде: нужно воспользоваться зависящими от версии комментариями, которые сервер воспринимает как потенциально исполняемый код (то есть код, который сервер будет выполнять, если номер его версии не меньше указанного). И сервер, и клиентские программы знают, что это не обычные комментарии, поэтому не удаляют их. Чтобы такой код гарантированно не выполнялся, можно просто задать очень большой номер версии, скажем `99999`. Добавим в триггер некоторые комментарии из предыдущего примера, чтобы сделать его более понятным:

```
CREATE TRIGGER fake_statement_trigger
BEFORE INSERT ON sometable
FOR EACH ROW
```

```
BEGIN
    DECLARE v_row_count INT DEFAULT ROW_COUNT();
    /*!99999      ROW_COUNT() равно 1 для всех строк, кроме первой,
      поэтому этот триггер выполняется один раз на всю команду. */
    IF v_row_count <> 1 THEN
       -- Здесь ваш код
    END IF;
END;
```

Курсоры

В настоящее время MySQL предлагает однонаправленные (с прокруткой вперед) серверные курсоры только для чтения, и использовать их можно лишь в хранимых процедурах на нижнем уровне клиента API. Курсоры допускают только чтение, потому что обходят временные таблицы, а не таблицы, в которых хранятся реальные данные. Курсор позволяет построчно обойти результат запроса, извлекая строки в переменные для последующей обработки. Хранимая процедура позволяет открывать сразу несколько курсоров, причем они могут быть вложены друг в друга.

Для непосвященного устройство курсоров в MySQL таит немало сюрпризов. Поскольку курсоры реализованы с помощью временных таблиц, у разработчика может возникнуть обманчивое ощущение эффективности. Самое важное, что нужно помнить, — то, что *курсор выполняет весь запрос в момент открытия*. Рассмотрим следующую процедуру:

```
1  CREATE PROCEDURE bad_cursor()
2  BEGIN
3     DECLARE film_id INT;
4     DECLARE f CURSOR FOR SELECT film_id FROM sakila.film;
5     OPEN f;
6     FETCH f INTO film_id;
7     CLOSE f;
8  END
```

Из этого примера видно, что курсор можно закрыть до завершения обхода результатов. Разработчик, привыкший к Oracle или Microsoft SQL Server, возможно, и не заметит в этой процедуре ничего плохого, но в MySQL она потребует массы лишней работы. Профилирование процедуры с помощью команды `SHOW STATUS` показывает, что она выполняет 1000 операций чтения индекса и 1000 операций вставки. Это происходит потому, что в таблице `sakila.film` ровно 1000 строк. Все 1000 операций чтения и записи производятся при обработке строки 5, еще до начала выполнения строки 6.

Мораль заключается в том, что раннее закрытие курсора, выбирающего данные из большой таблицы, не дает никакой экономии. Если нужно всего несколько строк, воспользуйтесь ключевым словом `LIMIT`.

Из-за курсоров MySQL может выполнять и дополнительные операции ввода/вывода, которые могут быть весьма медленными. Поскольку временные таблицы в памяти не поддерживают типы `BLOB` и `TEXT`, MySQL вынуждена создавать временные таблицы на диске для курсоров, извлекающих данные таких типов. Но даже если это не так,

временные таблицы, размер которых превышает значение параметра `tmp_table_size`, все равно создаются на диске.

MySQL не поддерживает курсоры на стороне клиента, но в клиентском API есть функции, которые эмулируют курсор путем загрузки всего результирующего набора в память. На самом деле это ничем не отличается от копирования результата в созданный приложением массив с последующей манипуляцией этим массивом. Дополнительная информация о последствиях для производительности загрузки всех результатов в память клиента содержится в главе 6.

Подготовленные операторы

Начиная с версии MySQL 4.1, сервер поддерживает *подготовленные операторы*, которые используют расширенный двоичный клиент-серверный протокол, обеспечивающий эффективную передачу данных между клиентом и сервером. Получить доступ к функциональности подготовленных операторов позволяют библиотеки, поддерживающие новый протокол, например MySQL C API. Библиотеки MySQL Connector/J и MySQL Connector/NET предлагают те же возможности для Java и .NET соответственно. Существует также SQL-интерфейс для подготовленных операторов, который мы рассмотрим в дальнейшем (он запутанный).

В момент создания подготовленного оператора клиентская библиотека посылает серверу прототип будущего запроса. Сервер разбирает и обрабатывает эту заготовку, сохраняет структуру, представляющую частично оптимизированный запрос, и возвращает клиенту *дескриптор команды*.

В подготовленных операторах могут быть параметры, обозначенные вопросительными знаками, вместо которых в момент выполнения подставляются фактические значения. Например, можно подготовить такой запрос:

```
INSERT INTO tbl(col1, col2, col3) VALUES (?, ?, ?);
```

Чтобы впоследствии выполнить этот запрос, серверу необходимо отправить дескриптор оператора и значения всех параметров, представленных вопросительными знаками. Это действие можно повторять сколько угодно раз. Способ отправки дескриптора оператора серверу зависит от языка программирования. Один из вариантов — воспользоваться MySQL-коннекторами для Java и .NET. Многие клиентские библиотеки, компонуемые с библиотеками MySQL на языке C, тоже предоставляют интерфейс к двоичному протоколу (вам следует ознакомиться с документацией по конкретному MySQL API).

Использование подготовленных операторов может оказаться эффективнее повторного выполнения запросов по нескольким причинам.

- Серверу нужно разобрать запрос только один раз.
- Сервер должен проделать некоторые шаги оптимизации однократно, так как он кэширует частичный план выполнения запроса.

- Отправка параметров в двоичном виде эффективнее передачи в виде ASCII-текста. Например, для отправки значения типа `DATE` нужно всего 3 байта вместо 10 при передаче в ASCII-виде. Но наибольшая экономия достигается для значений типа `BLOB` и `TEXT`, которые можно отправлять серверу блоками, а не одним гигантским куском. Таким образом, двоичный протокол позволяет экономить память клиента, а заодно уменьшает сетевой трафик и устраняет накладные расходы на преобразование данных из естественного формата хранения в недвоичную кодировку.
- Для каждого выполнения запроса нужно посылать только параметры, а не весь текст запроса, что также снижает сетевой трафик.
- MySQL сохраняет параметры прямо в буферах сервера, поэтому серверу не нужно копировать значения из одного места в памяти в другое.

Вдобавок подготовленные операторы повышают безопасность. Нет необходимости экранировать специальные символы на уровне приложения, что удобнее и делает программу менее уязвимой к внедрению SQL-кода или другим видам атак. (Никогда не стоит доверять данным, полученным от пользователей, даже при использовании подготовленных операторов.)

Двоичный протокол применим только к подготовленным операторам. При отправке запросов с помощью обычной функции API `mysql_query()` двоичный протокол не применяется. Многие клиентские библиотеки позволяют подготовить оператор, включив в его текст вопросительные знаки, а затем задавать фактические значения параметров при каждом выполнении. Но зачастую это всего лишь эмуляция цикла «подготовка — отправка» в клиентском коде, а реально каждый запрос отправляется серверу с помощью `mysql_query()`.

Оптимизация подготовленных операторов

MySQL кэширует частичные планы выполнения для подготовленных операторов, но некоторые оптимизации зависят от фактических значений параметров и поэтому не могут быть заранее вычислены и закэшированы. Все оптимизации можно разделить на три типа в зависимости от того, когда они выполняются. На момент написания книги действовала следующая классификация.

- *На этапе подготовки* сервер разбирает текст запроса, устраняет отрицания и переписывает подзапросы.
- *При первом выполнении* сервер упрощает вложенные соединения и преобразует `OUTER JOIN` в `INNER JOIN` там, где это возможно.
- *При каждом выполнении* сервер:
  - отсекает секции;
  - везде, где это возможно, исключает `COUNT()`, `MIN()` и `MAX()`;
  - устраняет константные подвыражения;
  - обнаруживает константные таблицы;

- распространяет равенство;
- анализирует и оптимизирует методы доступа `ref`, `range` и `index_ merge`;
- оптимизирует порядок соединения таблиц.

Более подробно обо всех этих оптимизациях мы поговорим в главе 6. Хотя некоторые из них теоретически можно выполнить только один раз, они все еще работают.

SQL-интерфейс для подготовленных операторов

SQL-интерфейс для подготовленных операторов существует начиная с версии 4.1. Он позволяет давать серверу инструкции о создании и выполнении подготовленных операторов, но использует не двоичный протокол. Вот пример использования подготовленного оператора из SQL:

```
mysql> SET @sql := 'SELECT actor_id, first_name, last_name
    -> FROM sakila.actor WHERE first_name = ?';
mysql> PREPARE stmt_fetch_actor FROM @sql;
mysql> SET @actor_name := 'Penelope';
mysql> EXECUTE stmt_fetch_actor USING @actor_name;
+----------+------------+-----------+
| actor_id | first_name | last_name |
+----------+------------+-----------+
|        1 | PENELOPE   | GUINESS   |
|       54 | PENELOPE   | PINKETT   |
|      104 | PENELOPE   | CRONYN    |
|      120 | PENELOPE   | MONROE    |
+----------+------------+-----------+
mysql> DEALLOCATE PREPARE stmt_fetch_actor;
```

Получив такие операторы, сервер транслирует их в те же самые операции, которые получились бы при использовании клиентской библиотеки. Поэтому никакого особого двоичного протокола для создания и выполнения подготовленных операторов вам применять не придется.

Как легко заметить, подобный синтаксис несколько тяжеловеснее, чем при вводе обычных команд `SELECT`. Тогда в чем же преимущество такого использования подготовленных операторов?

Основное применение они находят в хранимых процедурах. В MySQL 5.0 подготовленные операторы разрешено использовать внутри хранимых процедур, а их синтаксис аналогичен SQL-интерфейсу. Это означает, что в хранимых процедурах можно строить и выполнять динамические SQL-команды путем конкатенации строк, что еще больше повышает гибкость подобных конструкций. Далее приведен пример хранимой процедуры, в которой для каждой таблицы в указанной базе данных выполняется команда `OPTIMIZE TABLE`:

```
DROP PROCEDURE IF EXISTS optimize_tables;
DELIMITER //
CREATE PROCEDURE optimize_tables(db_name VARCHAR(64))
BEGIN
    DECLARE t VARCHAR(64);
    DECLARE done INT DEFAULT 0;
```

```
    DECLARE c CURSOR FOR
       SELECT table_name FROM INFORMATION_SCHEMA.TABLES
       WHERE TABLE_SCHEMA = db_name AND TABLE_TYPE = 'BASE TABLE';
    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
    OPEN c;
    tables_loop: LOOP
       FETCH c INTO t;
       IF done THEN
          LEAVE tables_loop;
       END IF;
       SET @stmt_text := CONCAT("OPTIMIZE TABLE ", db_name, ".", t);
       PREPARE stmt FROM @stmt_text;
       EXECUTE stmt;
          DEALLOCATE PREPARE stmt;
    END LOOP;
    CLOSE c;
 END//
 DELIMITER ;
```

Эту хранимую процедуру можно использовать следующим образом:

```
mysql> CALL optimize_tables('sakila');
```

Есть и другой способ написать цикл в процедуре:

```
REPEAT
    FETCH c INTO t;
    IF NOT done THEN
       SET @stmt_text := CONCAT("OPTIMIZE TABLE ", db_name, ".", t);
       PREPARE stmt FROM @stmt_text;
       EXECUTE stmt;
       DEALLOCATE PREPARE stmt;
    END IF;
UNTIL done END REPEAT;
```

Между этими двумя конструкциями цикла имеется важное различие: в случае `REPEAT` условие цикла проверяется дважды на каждой итерации. В данной ситуации это, пожалуй, не вызовет больших проблем с производительностью, так как мы просто сравниваем целочисленные значения, но более сложная проверка может стать и более затратной.

Имена таблиц и баз данных, созданные с помощью конкатенации строк, — это отличный пример применения SQL-интерфейса для подготовленных операторов, поскольку так можно строить директивы, не принимающие параметров. Невозможно подставить параметры вместо имен таблиц и баз данных, так как они являются идентификаторами. Еще одно возможное применение — конструирование раздела `LIMIT`, который тоже не допускает параметризации.

SQL-интерфейс хорош для ручного тестирования подготовленных операторов, но вне хранимых процедур он не особо полезен. Поскольку интерфейс основан на SQL, двоичный протокол не используется, а сетевой трафик не сокращается, так как при наличии параметров приходится отправлять дополнительные запросы, чтобы задать их значения. Выгоду удается получить в некоторых частных случаях, например при подготовке очень длинной строки SQL-запроса, который будет выполняться многократно без параметров.

Ограничения подготовленных операторов

Есть несколько ограничений и оговорок относительно применения подготовленных операторов.

- Подготовленные операторы локальны по отношению к соединению, поэтому в другом соединении тот же самый дескриптор использовать нельзя. По той же причине клиент, который разрывает и вновь устанавливает соединение, теряет все подготовленные операторы (смягчить эту проблему позволяют пул соединений и устойчивые соединения).
- До версии MySQL 5.1 подготовленные операторы не использовали кэш запросов.
- Применение подготовленных операторов не всегда эффективно. Если подготовленный оператор выполняется всего один раз, вы можете потратить на подготовку больше времени, чем ушло бы на выполнение обычной SQL-команды. Кроме того, для подготовки оператора необходимо дополнительное обращение к серверу (правильное использование подготовленного оператора предполагает освобождение их дескрипторов после применения).
- В настоящее время подготовленные операторы нельзя использовать в хранимых функциях, но можно — в хранимых процедурах.
- Если вы забудете освободить дескриптор подготовленного оператора, то возникнет утечка. Это может привести к потере большого количества ресурсов сервера. Кроме того, поскольку существует глобальное ограничение на количество подготовленных операторов, такая ошибка может отразиться на других соединениях — они не смогут подготовить оператор.
- Некоторые команды, такие как `BEGIN`, не могут выполняться в подготовленных операторах.

Однако, скорее всего, самым большим ограничением подготовленных операторов является тот факт, что легко запутаться в том, что они собой представляют и как работают. Иногда очень сложно объяснить разницу между тремя способами применения подготовленных операторов.

- *Эмуляция на стороне клиента.* Клиентский драйвер принимает строку с заполнителями, затем заменяет параметры в SQL и отправляет результирующий запрос на сервер.
- *Серверная сторона.* Драйвер отправляет строку с заполнителями на сервер специальным двоичным протоколом, получает обратно идентификатор оператора, затем выполняет оператор по двоичному протоколу, указывая идентификатор и параметры.
- *Интерфейс SQL.* Клиент отправляет строку с заполнителями на сервер в качестве оператора `PREPARE` SQL, устанавливает переменные SQL в значения параметров и, наконец, выполняет оператор с помощью команды `EXECUTE SQL`. Все это происходит через обычный текстовый протокол.

Функции, определяемые пользователем

MySQL уже давно поддерживает *функции, определяемые пользователем* (user-defined functions, UDF). В отличие от хранимых функций, которые создаются на языке SQL, функции, определяемые пользователем, можно писать на любом языке программирования, который поддерживает соглашения о вызове, принятые в языке C.

UDF необходимо сначала скомпилировать, а затем динамически скомпоновать с сервером. Поэтому они платформенно-зависимы, зато наделяют разработчика огромной мощью. Такие функции могут работать очень быстро и пользоваться необозримой функциональностью, которую предлагают операционная система и доступные библиотеки. Хранимые функции, написанные на SQL, хороши для простых операций, например для вычисления расстояния по дуге большого круга между двумя точками на глобусе, но если требуется отправить какие-то пакеты по сети, то без UDF не обойтись. Кроме того, в настоящий момент нельзя написать агрегатные функции как хранимые функции, а с помощью UDF это легко сделать.

Но большая власть накладывает большую ответственность. Ошибка в UDF может привести к аварийному останову сервера, повредить память или данные пользователя и вообще внести хаос, как любой плохо написанный код на C.

В отличие от хранимых функций, написанных на SQL, UDF пока не умеют читать из таблиц и писать в них, по крайней мере не в том же транзакционном контексте, в котором выполняется вызвавшая их команда. Это означает, что они полезны скорее для чистых вычислений или взаимодействия с внешним миром. MySQL постоянно расширяет возможности обращения к ресурсам, внешним по отношению к серверу. Отличным примером того, что можно сделать с помощью UDF, являются написанные Брайаном Эйкером (Brian Aker) и Патриком Гэлбрайтом (Patrick Galbraith) функции для взаимодействия с сервером memcached (http://tangent.org/586/Memcached_Functions_for_MySQL.html).

Если вы используете UDF, внимательно следите за изменениями в новых версиях MySQL, поскольку не исключено, что код придется перекомпилировать или даже модифицировать, чтобы он мог работать с новым сервером. Кроме того, UDF должны быть потокобезопасны, так как исполняются внутри сервера MySQL, то есть в многопоточном окружении.

Для MySQL есть много хороших библиотек с готовыми UDF и приведено немало примеров, показывающих, как их реализовать самостоятельно. Самый большой репозиторий UDF находится по адресу http://www.mysqludf.org.

Приведем код UDF `NOW_USEC()`, который будем использовать для измерения скорости репликации в главе 10:

```
#include <my_global.h>
#include <my_sys.h>
```

```
#include <mysql.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>
extern "C" {
    my_bool now_usec_init(UDF_INIT *initid, UDF_ARGS *args, char *message);
    char *now_usec(
                   UDF_INIT *initid,
                   UDF_ARGS *args,
                   char *result,
                   unsigned long *length,
                   char *is_null,
                   char *error);
}
my_bool now_usec_init(UDF_INIT *initid, UDF_ARGS *args, char *message) {
    return 0;
}
char *now_usec(UDF_INIT *initid, UDF_ARGS *args, char *result,
               unsigned long *length, char *is_null, char *error) {
  struct timeval tv;
  struct tm* ptm;
  char time_string[20]; /* e.g. "2006-04-27 17:10:52" */
  char *usec_time_string = result;
  time_t t;
  /* Получаем время суток и преобразуем его в структуру tm. */
  gettimeofday (&tv, NULL);
  t = (time_t)tv.tv_sec;

ptm = localtime (&t);
  /* Форматируем дату и время с точностью до секунд. */
  strftime (time_string, sizeof (time_string), "%Y-%m-%d %H:%M:%S", ptm);
  /* Выводим отформатированное время в виде секунд, за которыми идет
   * десятичная точка, а затем выводим микросекунды. */
  sprintf(usec_time_string, "%s.%06ld\n", time_string, tv.tv_usec);
  *length = 26;
  return(usec_time_string);
}
```

Одним из примеров использования функции, определенной пользователем, для решения сложной проблемы является кейс, рассмотренный в предыдущей главе. Мы также написали несколько UDF, одна из которых поставляется с пакетом Percona Toolkit и применяется для эффективного сбора контрольных сумм данных, поэтому вы можете протестировать целостность репликации с меньшими затратами, а другая — для предварительной обработки текста перед индексированием его для поиска с помощью Sphinx. UDF могут быть весьма мощными.

Плагины

В дополнение к UDF MySQL поддерживает множество плагинов. Они могут добавлять собственные параметры командной строки и переменные состояния, предоставлять таблицы `INFORMATION_SCHEMA`, запускаться как служебные процессы и многое другое. В MySQL версии 5.1 и последующих сервер имеет гораздо больше API-плагинов,

чем прежде, и теперь он может быть расширен множеством способов без изменения исходного кода. Приведем краткий список плагинов.

- *Плагины процедур.* Могут обрабатывать результирующий набор. Это довольно старый тип плагина, похожий на UDF, большинство людей даже не подозревают о его существовании и никогда не используют. Примером может служить встроенный плагин PROCEDURE ANALYZE.
- *Плагины служебных процессов.* Запускаются как процесс в MySQL и могут, например, прослушивать сетевые порты или выполнять периодические задания. Примером может служить плагин Handler-Socket, входящий в состав Percona Server. Он открывает сетевые порты и принимает простой протокол, который позволяет получать доступ к таблицам InnoDB через интерфейс Handler без использования SQL, что делает его высокопроизводительным NoSQL-интерфейсом на сервере.
- *Плагины INFORMATION_SCHEMA.* Могут предоставлять произвольные таблицы `INFORMATION_SCHEMA`.
- *Плагины полнотекстовых анализаторов.* Обеспечивают перехват процессов чтения и разбиения документа на слова для индексирования, поэтому с их помощью вы можете индексировать документы PDF с учетом имен файлов. Кроме того, они могут стать частью процесса сопоставления во время выполнения запроса.
- *Плагины аудита.* Собирают события в определенных точках выполнения запроса, поэтому их можно использовать, например, как способ журналирования событий, происходящих на сервере.
- *Плагины аутентификации.* Могут работать на стороне как клиента, так и сервера для расширения диапазона доступных механизмов проверки подлинности на сервере, включая аутентификацию PAM и LDAP.

Для получения дополнительной информации изучите руководство по MySQL или прочитайте книгу Сергея Голубчика (Sergei Golubchik) и Эндрю Хатчингса (Andrew Hutchings) *MySQL 5.1 Plugin Development* (издательство Packt). Если вам нужен плагин, но вы не знаете, как его написать, вам могут помочь многие поставщики услуг, включая Monty Program, Open Query, Percona и SkySQL, которые предоставят в ваше распоряжение компетентных специалистов.

Кодировка и схемы упорядочения

Кодировка — это соответствие двоичного кода определенному набору символов. *Схема упорядочения* — это набор правил сортировки для конкретной кодировки. Начиная с версии MySQL 4.1, с каждым символьным значением могут быть связаны кодировка и схема упорядочения[1]. Поддержка кодировок и схем упорядочения в MySQL реализована на уровне лучших мировых стандартов, но за нее приходится

[1] В MySQL 4.0 и более ранних версиях использовалась глобальная настройка для всего сервера и можно было выбрать одну из нескольких 8-разрядных кодировок.

расплачиваться дополнительной сложностью, а иногда и падением производительности. (Кстати, Drizzle отказался от любых кодировок, кроме UTF-8.)

В этом разделе объясняются настройки и функции, которых в большинстве случаев достаточно. Если вам интересны малоизвестные детали, почитайте руководство по MySQL.

Использование кодировок в MySQL

С каждой кодировкой может быть связано несколько схем упорядочения, и одна из них является схемой по умолчанию. Схема упорядочения принадлежит конкретной кодировке, и ни с какой другой ее использовать нельзя. Кодировка и схема упорядочения применяются совместно, поэтому отныне мы будем называть эту пару просто кодировкой.

В MySQL есть множество параметров для управления кодировками. Их легко спутать с самими кодировками, поэтому всегда помните: только символьные значения могут иметь кодировку. Во всех остальных случаях речь идет всего лишь о параметре, который говорит, какую кодировку использовать для сравнения и других операций. Символьным значением может быть значение, хранящееся в столбце, литерал в запросе, результат выражения, пользовательская переменная и т. д.

Параметры MySQL можно разделить на две категории: параметры, используемые по умолчанию при создании объектов, и параметры, управляющие взаимодействием между клиентом и сервером.

Параметры, используемые по умолчанию при создании объектов

В MySQL у сервера, каждой базы данных и каждой таблицы есть свои кодировка и схема упорядочения, применяемые по умолчанию. Они образуют иерархию умолчаний, на основе которой выбирается кодировка вновь создаваемого столбца. Это, в свою очередь, указывает серверу, в какой кодировке хранить значения в этом столбце.

На каждом уровне иерархии можно либо определить кодировку явно, либо позволить серверу использовать подходящие из применяемых по умолчанию.

- При создании базы данных кодировка наследуется от определенного на уровне сервера параметра `character_set_server`.
- При создании таблицы кодировка наследуется от базы данных.
- При создании столбца кодировка наследуется от таблицы.

Помните, что значения хранятся только в таблицах, поэтому на более высоких уровнях иерархии определены всего лишь применяемые по умолчанию. Кодировка, используемая по умолчанию для таблицы, никак не влияет на то, как хранятся значения в этой таблице, — это лишь способ сообщить MySQL, какую кодировку следует использовать при создании нового столбца, если она не указана явно.

Параметры взаимодействия между клиентом и сервером

Когда клиент и сервер взаимодействуют друг с другом, они могут посылать друг другу данные в разных кодировках. Сервер выполняет преобразование по мере необходимости.

- Сервер предполагает, что клиент посылает команды в кодировке, заданной параметром `character_set_client`.
- Получив команду от клиента, сервер преобразует ее в кодировку, заданную параметром `character_set_connection`. Этот же параметр управляет преобразованием чисел в строки.
- Результаты и сообщения об ошибках сервер преобразует в кодировку, заданную параметром `character_set_result`.

Этот процесс показан на рис. 7.2.

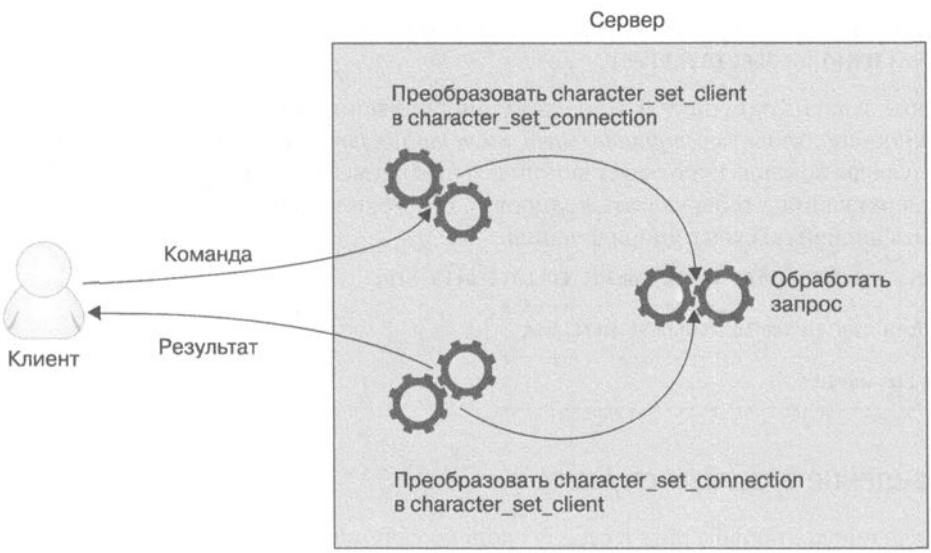


Рис. 7.2. Кодировки клиента и сервера

Изменить эти три параметра можно с помощью команд `SET NAMES` и/или `SET CHARACTER SET`. Отметим, однако, что и та и другая влияют *только на параметры сервера*. Клиентское приложение и API клиента должны быть правильно настроены, чтобы не возникало проблем при взаимодействии с сервером.

Предположим, что вы открываете клиентское соединение с кодировкой `latin1` (применяется по умолчанию, если не была изменена с помощью функции `mysql_options()`), а затем выполняете команду `SET NAMES utf8`, сообщая серверу о том, что клиент будет посылать данные в кодировке UTF-8. В результате получается несоответствие кодировок, которое может привести к ошибкам и даже поставить под угрозу безопасность. Необходимо установить кодировку клиента и использовать функцию

`mysql_real_escape_string()` для экранирования значений. В PHP для изменения кодировки клиента можно воспользоваться функцией `mysql_set_charset()`.

Как MySQL сравнивает значения

При сравнении двух значений в разных кодировках MySQL должна сначала привести их к общей кодировке. Если кодировки несовместимы, то возникнет ошибка `ERROR 1267 (HY000): Illegal mix of collations`. В этом случае следует воспользоваться функцией `CONVERT()` и явно преобразовать одно из значений в кодировку, совместимую с кодировкой другого значения. Начиная с версии MySQL 5.0, такое преобразование часто выполняется неявно, поэтому вышеупомянутая ошибка более характерна для MySQL 4.1.

MySQL также назначает значениям характеристику, которая называется *приводимостью*. Она определяет приоритет кодировки значения и влияет на то, какое значение MySQL будет неявно преобразовывать. Для отладки ошибок, связанных с кодировками и схемами упорядочения, можно использовать функции `CHARSET()`, `COLLATION()` и `COERCIBILITY()`.

Чтобы задать кодировку и/или схему упорядочения литералов в SQL-командах, можно использовать *вступительный элемент* и *ключевое слово* `collate`. Например, следующая команда использует вступительный элемент, которому предшествует знак подчеркивания, чтобы указать кодировку UTF-8, и ключевое слово `collate`, чтобы задать двоичную схему упорядочения:

```
mysql> SELECT _utf8 'hello world' COLLATE utf8_bin;
+-------------------------------------+
| _utf8 'hello world' COLLATE utf8_bin |
+-------------------------------------+
| hello world                         |
+-------------------------------------+
```

Поведение в особых случаях

Поведение кодировок таит в себе несколько сюрпризов. Далее описаны ситуации, о которых стоит помнить.

- Магический параметр `character_set_database`. По умолчанию параметр `character_set_database` совпадает с кодировкой базы данных, принимаемой по умолчанию. Когда изменяется база данных по умолчанию, изменяется и этот параметр. При подключении к серверу, для которого не определена база данных по умолчанию, он принимает значение `character_set_server`.
- `LOAD DATA INFILE`. Команда `LOAD DATA INFILE` интерпретирует входные данные в соответствии с текущим значением параметра `character_set_database`. MySQL 5.0 и более поздние версии принимают в данной команде необязательный параметр `CHARACTER SET`, но полагаться на это не стоит. Мы выяснили, что надежнее всего выбрать нужную базу данных с помощью команды `USE`, затем задать кодировку с помощью команды `SET NAMES`, а уже потом загружать данные. MySQL считает, что

все загружаемые данные записаны в одной и той же кодировке вне зависимости от того, какая кодировка задана для столбцов, в которые данные загружаются.

- `SELECT INTO OUTFILE`. Команда `SELECT INTO OUTFILE` выводит данные без перекодировки. В настоящее время нет другого способа задать данным кодировку, кроме как обернуть каждый столбец функцией `CONVERT()`.
- Внутренние экранированные последовательности. MySQL интерпретирует экранированные последовательности в командах в соответствии с параметром `character_set_client`, даже если имеется вступительный элемент или ключевое слово `COLLATE`. Это объясняется тем, что экранированные последовательности в литералах интерпретирует синтаксический анализатор, а он ничего не знает о схемах упорядочения. С точки зрения анализатора вступительный элемент не команда, а просто лексема.

Выбор кодировки и схемы упорядочения

Начиная с версии 4.1, MySQL поддерживает большой диапазон кодировок и схем упорядочения, в том числе Unicode с многобайтовыми символами UTF-8 (MySQL поддерживает трехбайтовое подмножество UTF-8, достаточное для представления практически всех символов в большинстве языков). Узнать, какие кодировки поддерживаются, можно с помощью команд `SHOW CHARACTER SET` и `SHOW COLLATION`.

Будьте проще

Смесь разных кодировок в одной базе данных может вызвать хаос. Несовместимые кодировки служат источником разнообразных ошибок. Все может работать хорошо, пока в данных не встретится какой-то конкретный символ, а потом начнутся сложности при выполнении тех или иных операций, например при соединении таблиц. Для разрешения проблемы приходится либо выполнять команду `ALTER TABLE`, чтобы преобразовать столбцы в совместимые кодировки, либо выполнять приведение к нужной кодировке с помощью вступительного элемента и ключевого слова `collate` в SQL-команде.

Чтобы избежать всех этих трудностей, стоит задать разумные умолчания на уровне сервера и, возможно, на уровне базы данных. А в исключительных случаях задавать кодировку на уровне столбца.

При выборе схемы упорядочения обычно исходят из того, как сортировать буквы: с учетом регистра, без учета регистра или в соответствии с двоичным кодом. Соответственно, названия схем упорядочения, как правило, заканчиваются на `_cs`, `_ci` или `bin`, чтобы проще было определить, к какой их трех групп они относятся. Разница между схемами упорядочения с учетом регистра и двоичной заключается в том, что при схеме упорядочения с двоичным кодом сортировка выполняется в соответствии с байтовыми значениями символов, тогда как зависящие от регистра схемы

упорядочения могут иметь сложные правила сортировки, учитывающие несколько символов в языках, таких как, например, немецкий.

Когда кодировка задается явно, не обязательно указывать и имя кодировки, и имя схемы упорядочения. Если одно или оба имени опущены, MySQL подставит недостающее по умолчанию. В табл. 7.2 показано, как MySQL выбирает кодировки и схемы упорядочения.

Таблица 7.2. Как MySQL определяет кодировку и схему упорядочения по умолчанию

Если заданы	Выбранная кодировка	Выбранная схема упорядочения
Кодировка и схема упорядочения	Заданная	Заданная
Только кодировка	Заданная	По умолчанию для заданной кодировки
Только схема упорядочения	Та, к которой относится схема упорядочения	Заданная
Ни то ни другое	Применимое умолчание	Применимое умолчание

Следующие команды показывают, как создать базу данных, таблицу и столбец с явно заданными кодировкой и схемой упорядочения:

```
CREATE DATABASE d CHARSET latin1;
CREATE TABLE d.t(
    col1 CHAR(1),
    col2 CHAR(1) CHARSET utf8,
    col3 CHAR(1) COLLATE latin1_bin
) DEFAULT CHARSET=cp1251;
```

В получившейся таблице для столбцов будут действовать следующие схемы упорядочения:

```
mysql> SHOW FULL COLUMNS FROM d.t;
+------+---------+-------------------+
|Field | Type    | Collation         |
+------+---------+-------------------+
|col1  | char(1) | cp1251_general_ci |
|col2  | char(1) | utf8_general_ci   |
|col3  | char(1) | latin1_bin        |
+------+---------+-------------------+
```

Как кодировка и схема упорядочения влияют на запросы

При использовании некоторых кодировок может возрасти потребление ресурсов процессора, памяти и места на диске. Возможно, даже не удастся воспользоваться индексами. Поэтому выбирать кодировку и схему упорядочения следует очень тщательно.

Преобразование из одной кодировки или схемы упорядочения в другую может повлечь за собой накладные расходы при выполнении некоторых операций. Так, по

столбцу `title` таблицы `sakila.film` построен индекс, способный ускорить выполнение запросов с фразой `ORDER BY`:

```
mysql> EXPLAIN SELECT title, release_year FROM sakila.film ORDER BY title\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: index
possible_keys: NULL
          key: idx_title
      key_len: 767
          ref: NULL
         rows: 953
        Extra:
```

Однако сервер может использовать этот индекс для сортировки, только если он обработан в соответствии с той же схемой упорядочения, которая указана в запросе. При построении индекса используется схема упорядочения столбца, в данном случае `utf8_general_ci`. Если требуется отсортировать результаты на основе другой схемы упорядочения, то сервер прибегнет к файловой сортировке:

```
mysql> EXPLAIN SELECT title, release_year
    -> FROM sakila.film ORDER BY title COLLATE utf8_bin\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 953
        Extra: Using filesort
```

MySQL должна не только адаптироваться к кодировке, заданной по умолчанию для соединения, а также ко всем параметрам, явно указанным в запросах, но и выполнять перекодирование, чтобы можно было сравнивать значения, записанные в разных кодировках. Например, при соединении двух таблиц по столбцам, имеющим разные кодировки, MySQL вынуждена перекодировать один из них. В результате может получиться так, что не удастся воспользоваться индексом, поскольку перекодировку можно уподобить функции, обертывающей столбец. Если вы не уверены, происходит ли что-то подобное, можете использовать `EXPLAIN EXTENDED`, а затем `SHOW WARNINGS`, чтобы просмотреть запрос с точки зрения сервера. Взглянув на кодировку в запросе, можно определить, проводилась ли перекодировка.

В случае применения многобайтовой кодировки UTF-8 для хранения символов отводится различное число байтов (от одного до трех). Для выполнения многих операций со строками MySQL применяет буферы фиксированного размера, поэтому приходится выделять память, исходя из максимально возможной длины строки. Например, для хранения значения типа `CHAR(10)` в кодировке UTF-8 потребуется 30 байт, даже если фактическая строка не содержит так называемых широких символов. При хранении

полей переменной длины (VARCHAR, TEXT) на диске эта проблема не возникает, но во временных таблицах в памяти, которые используются для обработки запросов и сортировки результатов, место всегда выделяется по максимальной длине.

В многобайтовых кодировках символ и байт не одно и то же. Поэтому в MySQL имеется две функции, LENGTH() и CHAR_LENGTH(), которые при использовании многобайтовой кодировки возвращают разные результаты. При работе с такой кодировкой для подсчета символов пользуйтесь функцией CHAR_LENGTH() (например, при вычислении параметров функции SUBSTRING()). То же предостережение действует и для многобайтовых кодировок в языках программирования приложений.

Еще один возможный сюрприз — ограничения на индексы. Если индексируется столбец в кодировке UTF-8, то MySQL вынуждена предполагать, что каждый символ может занимать до 3 байт, поэтому обычные ограничения на длину неожиданно уменьшаются в три раза:

```
mysql> CREATE TABLE big_string(str VARCHAR(500), KEY(str)) DEFAULT CHARSET=utf8;
Query OK, 0 rows affected, 1 warning (0.06 sec)
mysql> SHOW WARNINGS;
+---------+------+-------------------------------------------------------------+
| Level   | Code | Message                                                     |
+---------+------+-------------------------------------------------------------+
| Warning | 1071 | Specified key was too long; max key length is 999 bytes |
+---------+------+-------------------------------------------------------------+
```

Обратите внимание на то, что MySQL автоматически укорачивает индекс до 333-символьного префикса:

```
mysql> SHOW CREATE TABLE big_string\G
*************************** 1. row ***************************
       Table: big_string
Create Table: CREATE TABLE `big_string` (
  `str` varchar(500) default NULL,
  KEY `str` (`str`(333))
) ENGINE=MyISAM DEFAULT CHARSET=utf8
```

Если вы не обратите внимания на предупреждение и посмотрите на определение таблицы, то увидите, что индекс создан лишь по части столбца. При этом могут возникнуть нежелательные побочные эффекты, в частности, будет невозможно использовать покрывающие индексы.

Иногда рекомендуют всегда и везде использовать кодировку UTF-8, чтобы упростить себе жизнь. Однако с точки зрения производительности это далеко не всегда хорошо. Многим приложениям кодировка UTF-8 совсем ни к чему, а в зависимости от характера данных при ее использовании может потребоваться гораздо больше места на диске.

Принимая решение о выборе кодировки, нужно подумать о том, какие данные вы собираетесь хранить. Например, если речь идет преимущественно об англоязычном тексте, то кодировка UTF-8 практически не повлияет на используемое дисковое пространство, так как большинство символов английского языка кодируются одним байтом. Но для языков, алфавит которых отличается от латиницы, например русского или арабского, разница может оказаться очень заметной. Если приложение должно хранить только арабские символы, то можно воспользоваться кодировкой cp1256,

в которой все знаки арабского алфавита кодируются одним байтом. Но если нужно представлять тексты на разных языках и вы выбрали кодировку UTF-8, то для тех же самых арабских символов потребуется больше места. Аналогично преобразование столбца из национальной кодировки в UTF-8 может существенно увеличить объем занимаемого на диске места. При использовании InnoDB размер данных может увеличиться настолько, что значение перестанет помещаться на одной странице и придется задействовать внешнюю память, а это приводит к непроизводительному расходованию места на диске и фрагментации.

Иногда вообще не нужно использовать кодировку. Кодировки полезны прежде всего для сравнения с учетом регистра, сортировки и тех операций со строками, в которых нужно распознавать границы между символами, например `SUBSTRING()`. Если не требуется, чтобы сервер баз данных мог обрабатывать символьные значения, то можно хранить все, включая данные в кодировке UTF-8, в столбцах типа `BINARY`. При этом можно добавить специальный столбец, в котором будет содержаться информация о том, в какой кодировке представлена информация. Этот подход используется уже давно, однако он требует большой внимательности. Если забыть о том, что байт и символ не одно и то же, могут возникнуть трудные для отладки ошибки, например, при использовании функций `SUBSTRING()` и `LENGTH()`. Мы рекомендуем по возможности избегать такой практики.

Полнотекстовый поиск

В большинстве типичных запросов присутствует раздел `WHERE`, в котором значения сравниваются на равенство, выделяются диапазоны строк и т. д. Но иногда нужно искать по ключевому слову, и в этом случае поиск должен быть основан на релевантности, а не на простом сравнении строк. Для этой цели и предназначены системы полнотекстового поиска.

Для полнотекстового поиска требуется специальный синтаксис запроса. Индекс необязателен, но при его наличии поиск выполняется быстрее. Полнотекстовые индексы имеют специальную структуру, ускоряющую поиск документов, содержащих заданные ключевые слова.

По крайней мере с одним типом систем полнотекстового поиска вы точно знакомы, даже если не осознавали этого. Речь идет о поисковых системах в Интернете. Хотя работают они с гигантскими объемами данных и обычно не используют реляционные базы данных для их хранения, тем не менее основные принципы схожи.

Полнотекстовый поиск позволяет искать в символьных столбцах (типа `CHAR`, `VARCHAR` и `TEXT`) и обеспечивает как булев поиск, так и поиск на естественном языке. При реализации полнотекстового поиска действует целый ряд ограничений[1], кроме того, она

[1] В MySQL 5.1 можно использовать плагины полнотекстовых анализаторов для расширения полнотекстового поиска. Возможно, вы придете к выводу, что ограничения на полнотекстовый поиск в MySQL делают его использование в вашей программе бессмысленным. В приложении E мы обсудим поисковую систему Sphinx.

довольно сложна, но все же находит широкое применение, так как является частью сервера и пригодна для многих приложений. В этом разделе мы познакомимся с тем, как эту систему использовать и как проектировать полнотекстовый поиск с учетом производительности.

На момент написания книги в стандартной MySQL только подсистема хранения MyISAM поддерживает полнотекстовый поиск. Планируется реализация полнотекстового поиска в InnoDB в релизе MySQL 5.6. Кроме того, существуют сторонние подсистемы хранения, использующие полнотекстовый поиск, например Groonga.

То, что только MyISAM поддерживает полнотекстовый поиск, является серьезным ограничением, которое делает его неудачным решением для большинства приложений, поскольку слишком болезненно иметь дело с блокировкой на уровне таблиц, повреждением данных и восстановлением после сбоев. Как правило, вы можете просто использовать другое решение, такое как Sphinx, Lucene, Solr, Groonga, Xapian или Senna, или дождаться выхода MySQL 5.6 и использовать InnoDB. Тем не менее, если использование MyISAM подходит для вашего приложения, читайте дальше.

Полнотекстовый индекс MyISAM оперирует *полнотекстовым набором*, который составлен из одного или нескольких столбцов какой-либо таблицы. По сути дела, MySQL конкатенирует все столбцы, входящие в набор, и индексирует результат как одну длинную текстовую строку.

Полнотекстовый индекс MyISAM представляет собой специальный вид индекса, упорядоченного на основе B-дерева, с двумя уровнями. На первом уровне находятся ключевые слова. На втором уровне расположены списки ассоциированных со всеми ключевыми словами *указателей на документы*, ведущих на полнотекстовые наборы, в которых встречается данное ключевое слово. В индекс не добавляются все слова, из которых состоит набор, часть отбрасывается, а именно:

- слова, входящие в список *стоп-слов*, то есть шум. По умолчанию перечень стоп-слов построен на базе традиционного словоупотребления в английском языке, но параметр `ft_stopword_file` позволяет заменить его списком, взятым из внешнего файла;
- слова, содержащие меньше чем `ft_min_word_len` символов или больше чем `ft_max_word_len` символов.

В полнотекстовом индексе не хранится информация о том, в каком столбце набора находится ключевое слово, поэтому, если необходимо искать по различным комбинациям столбцов, придется создать несколько индексов.

Это также означает, что в разделе `MATCH AGAINST` нельзя сказать, будто слова, встречающиеся в одном столбце, важнее слов, встречающихся во всех остальных. А это типичное требование при построении систем поиска по сайтам. Например, иногда желательно, чтобы документы, в которых ключевое слово встречается в заголовке, стояли в списке результатов раньше остальных. Если для вас это существенно, то придется писать более сложные запросы (пример приведем далее).

Полнотекстовые запросы на естественном языке

Поисковый запрос на естественном языке определяет релевантность каждого документа исходному запросу. Релевантность вычисляется на основе количества совпавших слов и частоты их вхождения в документ. Чем реже слово встречается в индексе, тем более релевантным становится совпадение. И наоборот, слова, употребляемые очень часто, вообще не стоит искать. При полнотекстовом поиске на естественном языке исключаются слова, которые встречаются более чем в 50 % строк таблицы, даже если их нет в списке стоп-слов[1].

Синтаксис полнотекстового поиска несколько отличается от запросов других типов. Чтобы MySQL выполнила полнотекстовый поиск, в разделе `WHERE` должен присутствовать предикат `MATCH AGAINST`. Давайте рассмотрим пример. В демонстрационной базе Sakila по столбцам `title` и `description` таблицы `film_text` построен полнотекстовый индекс:

```
mysql> SHOW INDEX FROM sakila.film_text;
+-----------+----------------------+-------------+------------+
| Table     | Key_name             | Column_name | Index_type |
+-----------+----------------------+-------------+------------+
| ...
| film_text | idx_title_description | title       | FULLTEXT   |
| film_text | idx_title_description | description | FULLTEXT   |
+-----------+----------------------+-------------+------------+
```

Приведем пример полнотекстового запроса на естественном языке:

```
mysql> SELECT film_id, title, RIGHT(description, 25),
    ->    MATCH(title, description) AGAINST('factory casualties') AS relevance
    -> FROM sakila.film_text
    -> WHERE MATCH(title, description) AGAINST('factory casualties');
+---------+----------------------+--------------------------+----------------+
| film_id | title                | RIGHT(description, 25)   | relevance      |
+---------+----------------------+--------------------------+----------------+
|     831 | SPIRITED CASUALTIES  | a Car in A Baloon Factory | 8.4692449569702 |
|     126 | CASUALTIES ENCINO    | Face a Boy in A Monastery | 5.2615661621094 |
|     193 | CROSSROADS CASUALTIES | a Composer in The Outback | 5.2072987556458 |
|     369 | GOODFELLAS SALUTE    | d Cow in A Baloon Factory | 3.1522686481476 |
|     451 | IGBY MAKER           | a Dog in A Baloon Factory | 3.1522686481476 |
```

MySQL выполнила полнотекстовый поиск, разбив поисковую строку на слова и сопоставив каждое из них с содержимым полей `title` и `description`, которые были объединены в один полнотекстовый набор на этапе построения индекса. Обратите внимание на то, что только одна строка содержит оба этих слова и что три результата, содержащие слово `casualties` (таких всего три во всей таблице), перечислены в начале. Это объясняется тем, что в индексе результаты отсортированы по убыванию релевантности.

---

[1] При тестировании часто допускают ошибку: помещают несколько строк с тестовыми данными в полнотекстовый индекс, а потом обнаруживают, что ничего не найдено. Проблема в том, что каждое слово встречается более чем в половине строк таблицы.

В отличие от результатов обычных запросов результаты полнотекстового поиска автоматически сортируются по релевантности. При полнотекстовом запросе MySQL не может использовать для сортировки индекс. Поэтому, если хотите избежать файловой сортировки, не включайте в запрос раздел ORDER BY.

Как видно из примера, функция `MATCH()` возвращает показатель релевантности в виде числа с плавающей точкой. Это можно использовать для фильтрации результатов по релевантности или для показа релевантности в пользовательском интерфейсе. Функцию `MATCH()` можно употреблять более одного раза, не опасаясь дополнительных накладных расходов: MySQL понимает, что речь идет об одном и том же, и выполняет операцию только один раз. Однако если поместить вызов `MATCH()` в условие `ORDER BY`, то MySQL для упорядочения результатов прибегнет к файловой сортировке.

Столбцы в разделе `MATCH()` следует перечислять точно в том же порядке, в котором они были заданы при построении полнотекстового индекса. В противном случае MySQL не сможет воспользоваться индексом. Это происходит из-за того, что индекс не содержит сведений о том, в каком столбце встретилось ключевое слово.

Как мы уже отмечали, это также означает, что при полнотекстовом поиске нельзя сказать, что ключевое слово должно встречаться в конкретном столбце. Однако существует обходной путь: можно выполнить нестандартную сортировку с помощью нескольких полнотекстовых индексов по различным комбинациям столбцов, чтобы добиться необходимого ранжирования. Предположим, мы считаем столбец `title` более важным. Тогда можно добавить еще один индекс по этому столбцу:

```
mysql> ALTER TABLE film_text ADD FULLTEXT KEY(title) ;
```

Теперь столбец `title` можно сделать в два раза более важным для ранжирования:

```
mysql> SELECT film_id, RIGHT(description, 25),
    -> ROUND(MATCH(title, description) AGAINST('factory casualties'), 3)
    ->    AS full_rel,
    -> ROUND(MATCH(title) AGAINST('factory casualties'), 3) AS title_rel
    -> FROM sakila.film_text
    -> WHERE MATCH(title, description) AGAINST('factory casualties')
    -> ORDER BY (2 * MATCH(title) AGAINST('factory casualties'))
    ->    + MATCH(title, description) AGAINST('factory casualties') DESC;
+---------+--------------------------+----------+-----------+
| film_id | RIGHT(description, 25)   | full_rel | title_rel |
+---------+------------- ------------+----------+-----------+
|     831 | a Car in A Baloon Factory |    8.469 |     5.676 |
|     126 | Face a Boy in A Monastery |    5.262 |     5.676 |
|     299 | jack in The Sahara Desert |    3.056 |     6.751 |
|     193 | a Composer in The Outback |    5.207 |     5.676 |
|     369 | d Cow in A Baloon Factory |    3.152 |     0.000 |
|     451 | a Dog in A Baloon Factory |    3.152 |     0.000 |
|     595 | a Cat in A Baloon Factory |    3.152 |     0.000 |
|     649 | nizer in A Baloon Factory |    3.152 |     0.000 |
```

Однако из-за файловой сортировки такой подход, как правило, неэффективен.

Булев полнотекстовый поиск

При булевом поиске в самом запросе задается относительная релевантность каждого слова. Как и раньше, для фильтрации шума используется список стоп-слов, однако требование о том, чтобы слова были длиннее чем `ft_min_word_len` символов и короче чем `ft_max_word_len` символов, не предъявляется[1]. Результаты никак не сортируются.

При конструировании булева запроса можно использовать префиксы для относительного ранжирования каждого слова в поисковой строке. Наиболее часто употребляющиеся модификаторы приведены в табл. 7.3.

Таблица 7.3. Наиболее употребительные модификаторы для булева полнотекстового поиска

Пример	Значение
dinosaur	Строки, содержащие слово dinosaur, имеют больший ранг
~dinosaur	Строки, содержащие слово dinosaur, имеют меньший ранг
+dinosaur	Строка должна содержать слово dinosaur
-dinosaur	В строке не должно быть слова dinosaur
dino*	Строки, содержащие слова, которые начинаются с dino, имеют больший ранг

Можно использовать и другие операторы, например скобки для группировки. Таким образом можно конструировать сложные запросы.

В качестве примера еще раз рассмотрим таблицу `sakila.film_text` и найдем в ней фильмы, в описании которых содержатся слова `factory` и `casualties`. Как мы уже видели, поиск на естественном языке возвращает результаты, где есть одно или оба слова. Но если воспользоваться булевым поиском, то можно потребовать, чтобы обязательно встречались оба слова:

```
mysql> SELECT film_id, title, RIGHT(description, 25)
    -> FROM sakila.film_text
    -> WHERE MATCH(title, description)
    ->    AGAINST('+factory +casualties' IN BOOLEAN MODE);
+---------+--------------------+--------------------------+
| film_id | title              | RIGHT(description, 25)   |
+---------+--------------------+--------------------------+
|     831 | SPIRITED CASUALTIES | a Car in A Baloon Factory |
+---------+--------------------+--------------------------+
```

Можно также произвести *поиск фразы*, заключив несколько слов в кавычки. Это означает, что должна встречаться в точности указанная фраза:

```
mysql> SELECT film_id, title, RIGHT(description, 25)
    -> FROM sakila.film_text
```

[1] Полнотекстовые индексы даже не будут содержать слишком коротких или слишком длинных слов, но это другое дело. Здесь мы говорим о том, что сервер не будет удалять слова из поисковой фразы, если они слишком короткие или слишком длинные, что он обычно делает в процессе оптимизации запросов.

```
    -> WHERE MATCH(title, description)
    ->    AGAINST('"spirited casualties"' IN BOOLEAN MODE);
+---------+--------------------+--------------------------+
| film_id | title              | RIGHT(description, 25)   |
+---------+--------------------+--------------------------+
|     831 | SPIRITED CASUALTIES | a Car in A Baloon Factory |
+---------+--------------------+--------------------------+
```

Поиск фразы может выполняться довольно медленно. Один лишь поиск по полнотекстовому индексу не в состоянии дать ответ на такой запрос, поскольку индекс не содержит информации о том, как слова расположены друг относительно друга в исходном полнотекстовом наборе. Поэтому сервер должен анализировать сами строки.

Чтобы выполнить такой запрос, сервер сначала находит все документы, содержащие оба слова, spirited и casualties. Затем выбирает строки из найденных документов и проверяет вхождение фразы целиком. Поскольку на первом этапе используется полнотекстовый индекс, может сложиться впечатление, что поиск производится очень быстро, гораздо быстрее, чем с помощью эквивалентной операции `LIKE`. Это действительно так, если входящие во фразу слова не являются общеупотребительными, так что поиск по полнотекстовому индексу возвращает не слишком много документов. Если же эти слова встречаются очень часто, то `LIKE` может оказаться намного быстрее, так как в этом случае строки читаются последовательно, а не в квазислучайном порядке индекса, и обращаться к полнотекстовому индексу вовсе не требуется.

Для булева поиска полнотекстовый индекс фактически не нужен, хотя для него требуется подсистема хранения MyISAM. Если такой индекс есть, он просматривается, в противном случае сканируется вся таблица. Можно даже применить булев полнотекстовый поиск к столбцам из нескольких таблиц, например к результату соединения. Впрочем, во всех таких случаях процедура поиска будет выполняться медленно.

Изменения в полнотекстовом поиске в версии MySQL 5.1

В версии MySQL 5.1 полнотекстовый поиск претерпел некоторые изменения. Это и улучшенная производительность, и возможность подключать внешние анализаторы, расширяющие встроенные возможности. Например, подключаемый анализатор может изменить способ индексирования. С его помощью можно разбивать текст на слова более гибко, чем по умолчанию (скажем, стало возможно определить, что `"C++"` — это одно слово), выполнять первичную обработку, индексировать документы различных форматов (например, PDF) или реализовать пользовательский алгоритм морфологического поиска. Плагины могут также изменить способ выполнения поиска, например подвергнуть поисковые слова морфологическому поиску.

Компромиссы полнотекстового поиска и обходные пути

Реализация полнотекстового поиска в MySQL имеет несколько архитектурных ограничений, которые могут препятствовать решению конкретных задач. Однако существует ряд способов эти ограничения обойти.

Например, полнотекстовое индексирование в MySQL поддерживает единственную форму ранжирования по релевантности: частоту. В индексе не хранится расположение слова в строке, поэтому учитывать близость при вычислении релевантности нельзя. Во многих случаях (особенно если объем данных невелик) это не страшно, но вас такой подход может не устроить, а MySQL не позволяет выбрать другой алгоритм ранжирования (она даже не хранит данные, необходимые для ранжирования по близости).

Вторая проблема — размер. Полнотекстовое индексирование в MySQL работает хорошо, если весь индекс помещается в памяти, но если это не так, поиск может быть очень медленным, особенно когда поля велики. Для поиска по фразе приемлемая производительность достигается, когда и данные, и индексы находятся в памяти. Вставка строк в полнотекстовый индекс, его обновление и удаление строк из него намного более затратны, чем индексы других типов.

- Модификация фрагмента текста, содержащего 100 слов, требует не одной, а 100 операций с индексом.
- Скорость работы с другими типами индексов мало зависит от длины поля, а в случае полнотекстового индекса время индексирования текста из трех слов и из 10 000 слов различается на несколько порядков.
- Полнотекстовые индексы в гораздо большей степени подвержены фрагментации, поэтому выполнять команду `OPTIMIZE TABLE` приходится чаще.

Полнотекстовые индексы также влияют на оптимизацию запросов сервером. Выбор индекса, условия `WHERE` и `ORDER BY` работают не так, как можно было бы ожидать.

- Если имеется полнотекстовый индекс и в запросе присутствует фраза `MATCH AGAINST`, который может его использовать, то MySQL задействует его при обработке запроса. Она не станет сравнивать полнотекстовый индекс с другими индексами, которые можно было бы применить при выполнении запроса. Некоторые из таких индексов могли бы ускорить выполнение, но MySQL даже рассматривать их не будет.
- Полнотекстовый индекс пригоден только для выполнения полнотекстового поиска. Все остальные указанные в запросе (в частности, в разделе `WHERE`) условия нужно будет применять после считывания строки из таблицы. Это отличается от поведения индексов других типов, которые позволяют проверять сразу несколько условий в `WHERE`.
- В полнотекстовом индексе не хранится сам индексируемый текст, поэтому воспользоваться им как покрывающим не удастся.
- Полнотекстовые индексы можно использовать только для одного вида сортировки: по релевантности при поиске на естественном языке. Если нужно сортировать как-то иначе, MySQL прибегает к файловой сортировке.

Теперь посмотрим, как эти ограничения сказываются на запросах. Предположим, что имеется миллион документов, по которым сформированы обычный индекс по автору документа и полнотекстовый индекс по содержимому. Вы хотите выполнить

полнотекстовый поиск по содержимому, но только для документов, составленных автором 123. Возможно, вы напишете запрос следующим образом:

```
... WHERE MATCH(content) AGAINST ('High Performance MySQL')
    AND author = 123;
```

Однако он будет крайне неэффективен. Сначала MySQL осуществит полнотекстовый поиск по всему миллиону документов, так как отдает предпочтение полнотекстовому индексу. Затем применит раздел WHERE, чтобы оставить только документы указанного автора, но при этом не сможет воспользоваться индексом по автору.

Одно из решений описанной проблемы — включить идентификатор автора в полнотекстовый индекс. Можно выбрать какой-нибудь префикс, появление которого в тексте маловероятно, дописать после него идентификатор автора и включить это «слово» в столбец filters, который обновляется независимо (возможно, с помощью триггера).

Затем можно расширить полнотекстовый индекс, включив в него столбец filters, и переписать запрос так:

```
... WHERE MATCH(content, filters)
    AGAINST ('High Performance MySQL +author_id_123' IN BOOLEAN MODE);
```

Он может оказаться эффективнее, если идентификатор автора очень селективен, поскольку MySQL очень быстро сузит список документов, выполнив в полнотекстовом индексе поиск по слову author_id_123. Если же селективность невелика, то производительность может еще и ухудшиться. Поэтому не применяйте такой подход безоглядно.

Иногда полнотекстовые индексы можно использовать для поиска по ограничивающему прямоугольнику. Например, если вы хотите локализовать поиск некоторой прямоугольной областью на координатной плоскости (в случае географических приложений), то можно закодировать координаты в виде полнотекстового набора. Предположим, что в данной строке хранятся координаты $X = 123$ и $Y = 456$. Можно построить из них чередующуюся строку цифр XY142536 и поместить ее в столбец, включенный в полнотекстовый индекс. Теперь, если потребуется ограничить поиск, например, прямоугольником, для которого X изменяется от 10 до 199, а Y — от 400 до 499, то в запрос можно будет включить условие +XY14*. Это может оказаться гораздо быстрее, чем фильтрация с помощью раздела WHERE.

Еще один прием, в котором иногда удается с пользой применить полнотекстовые индексы, особенно для разбиения списка результатов на страницы, состоит в том, чтобы выбрать перечень первичных ключей полнотекстовым запросом и кэшировать результаты. Когда приложение будет готово к выводу итогов поиска, оно может отправить еще один запрос, отбирающий нужные строки по их идентификаторам. В этот запрос можно включить более сложные критерии или соединения, при обработке которых допустимо использовать другие индексы.

Полнотекстовые индексы поддерживаются только подсистемой хранения MyISAM, но если вы работаете с InnoDB или какой-то другой подсистемой, то можно реплицировать таблицы на подчиненный сервер, где они будут храниться в формате MyISAM

и обслуживать полнотекстовые запросы там. Если вы не хотите обрабатывать часть запросов на другом сервере, то можете секционировать таблицу по вертикали, отделив текстовые столбцы от прочих данных.

Можно также продублировать некоторые столбцы в таблицу, над которой построен полнотекстовый индекс. Эта стратегия применена к таблице `sakila.film_text`, которая поддерживается в актуальном состоянии с помощью триггеров. Еще одной альтернативой является использование внешней системой полнотекстового поиска, например Lucene или Sphinx. Подробнее о системе Sphinx можно прочитать в приложении Е.

Полнотекстовые запросы с фразой `GROUP BY` могут безнадежно «убить» производительность, поскольку полнотекстовый поиск обычно дает многочисленные результаты. Это приводит к произвольному вводу/выводу, а потом к построению временной таблицы или сортировке файлов для последующей группировки. Поскольку такие запросы часто применяются с целью поиска максимальных элементов в любой группе, то хорошим способом оптимизации может стать поиск по выборке из таблицы вместо стремления к абсолютной точности. Например, поместите первые 1000 строк во временную таблицу, а затем ищите максимальные элементы в каждой группе по этой выборке.

Настройка и оптимизация полнотекстового поиска

Одно из самых важных условий повышения производительности — регулярное обслуживание полнотекстовых индексов. Из-за большого количества слов в типичных документах и структуры двухуровневого B-дерева полнотекстовые индексы страдают от фрагментации больше, чем обычные. Для устранения фрагментации часто приходится выполнять команду `OPTIMIZE TABLE`. Если сервер выполняет много операций ввода/вывода, то гораздо быстрее будет периодически удалять и заново создавать полнотекстовые индексы.

Чтобы сервер эффективно производил полнотекстовый поиск, нужно выделить достаточно памяти под буферы ключей, чтобы полнотекстовые индексы целиком помещались в памяти, поскольку в этом случае они функционируют значительно лучше. Можно использовать отдельные буферы ключей, чтобы другие индексы не вытесняли полнотекстовый из памяти. Дополнительную информацию о буферах ключей MyISAM см. в главе 8.

Важно также создать качественный список стоп-слов. Список, предлагаемый по умолчанию, хорошо работает с англоязычной прозой, но для других языков или узкоспециализированных, в частности технических, текстов не годится. Например, при индексировании документа, относящегося к MySQL, имеет смысл сделать `mysql` стоп-словом, поскольку оно встречается слишком часто и потому бесполезно.

Часто удается повысить производительность за счет игнорирования коротких слов. Минимальная длина задается с помощью параметра `ft_min_word_len`. Если увеличить значение по умолчанию, то будет пропущено больше слов, поэтому индекс станет более коротким и быстрым, правда, менее точным. Не забывайте, впрочем, что для каких-то специальных целей могут быть значимы и очень короткие слова. Например,

если искать по запросу cd player в документах о бытовой электронике, то при запрете на короткие слова может быть выдано много нерелевантных результатов. Пользователь вряд ли хочет видеть документы о MP3- и DVD-плеерах, но если минимальная длина слова составляет четыре символа, то будет выполнен поиск только по слову player и, как следствие, возвращены документы, касающиеся вообще всех плееров.

Задание списка стоп-слов и минимальной длины запроса может за счет исключения некоторых слов из индекса ускорить поиск, но его качество при этом пострадает. Подходящий баланс зависит от приложения. Если требуется хорошая производительность и высокое качество поиска, то придется настроить оба параметра. Было бы полезно интегрировать в приложение какой-нибудь механизм журналирования и исследовать типичный поиск, нетипичный поиск, поиск, не возвращающий никаких результатов, и поиск, возвращающий очень много результатов. Таким образом можно получить полезную информацию о пользователях приложения и характере содержимого своих документов, а затем воспользоваться ею для повышения скорости и качества поиска.

Имейте в виду, что после изменения минимальной длины слова вам придется перестроить индекс командой OPTIMIZE TABLE, чтобы новое значение вступило в силу. Существует также параметр ft_max_word_len, предохраняющий от индексирования слишком длинных слов.

Если вы импортируете много данных в таблицы, содержащие проиндексированные текстовые столбцы, то перед началом импорта отключите полнотекстовые индексы командой `DISABLE KEYS`, а по завершении включите командой `ENABLE KEYS`. Обычно это дает существенный выигрыш во времени загрузки из-за высоких затрат на обновление индекса для каждой строки. А в качестве бонуса вы получите еще и дефрагментированный индекс.

Если набор данных очень велик, то имеет смысл провести секционирование вручную, распределив информацию по нескольким узлам и выполняя поиск на них параллельно. Это сложная задача, поэтому, возможно, стоит воспользоваться внешней системой полнотекстового поиска, например Lucene или Sphinx. Наш опыт показывает, что они могут обеспечить производительность на несколько порядков выше.

Распределенные транзакции

Если транзакции подсистем хранения (см. раздел «Транзакции» главы 1) обеспечивают свойства ACID внутри самой подсистемы, то распределенная (XA) транзакция, транзакция высокого уровня, позволяет распространить некоторые из этих свойств вовне подсистемы хранения и даже вовне базы данных — посредством механизма двухфазной фиксации. Начиная с версии 5.0 в MySQL реализована частичная поддержка XA-транзакций.

Для XA-транзакции требуется координатор транзакции, который просит всех участников подготовиться к коммиту (фаза 1). Получив от всех участников ответ «готов», координатор предлагает им выполнить коммит (фаза 2). MySQL может выступать в роли участника XA-транзакции, но не в роли координатора.

На самом деле в MySQL существует два вида XA-транзакций. Сервер MySQL может участвовать в любой управляемой извне распределенной транзакции, но он использует тот же механизм и внутри себя для координации подсистем хранения и двоичного журналирования.

Внутренние XA-транзакции

Причиной внутреннего использования XA-транзакций в MySQL является желание архитектурно отделить сервер от подсистем хранения. Подсистемы хранения абсолютно независимы друг от друга и ничего друг о друге не знают, поэтому любая транзакция, пересекающая границы подсистем, по своей природе является распределенной и нуждается в третьей стороне для координации. Этой третьей стороной является сервер MySQL. Не будь XA-транзакций, для коммита транзакции, пересекающей границы подсистем, пришлось бы последовательно просить каждую подсистему выполнить свою часть коммита. Но тогда аварийный останов в момент после того, как одна подсистема выполнила коммит, а другая еще не успела, привел бы к нарушению правил транзакционности (напомним, что транзакция по определению выполняет все или ничего).

Если взглянуть на двоичный журнал как на подсистему хранения журналированных событий, то становится понятно, почему XA-транзакции нужны даже тогда, когда в обработке запроса участвует лишь одна подсистема. Синхронизация коммита в смысле подсистемы хранения с коммитом события в двоичном журнале — это распределенная транзакция, поскольку за двоичный журнал отвечает сервер.

В настоящее время протокол XA создает некую дилемму в плане производительности. В версии 5.0 он сломал поддержку *группового коммита* (метод, позволяющий выполнить коммит для нескольких транзакций одной операцией ввода/вывода) в InnoDB, поэтому требуется выполнять больше системных вызовов `fsync()`, чем необходимо[1]. Кроме того, если двоичный журнал включен, то каждая транзакция должна синхронизироваться с двоичным журналом и вместо одного сброса в журнал на операцию коммита необходимо выполнить два. Другими словами, если требуется, чтобы двоичный журнал безопасно синхронизировался с транзакциями, то на каждую транзакцию будет приходиться по меньшей мере три вызова `fsync()`.

[1] На момент написания этой книги проводилась большая работа по исправлению проблемы с групповым коммитом. В результате получились по крайней мере три конкурирующие реализации. Остается узнать, какая из них попадает в официальный код MySQL, который будет использоваться большинством людей, или какая версия будет исправлена. Версия, доступная в MariaDB и Percona Server, кажется хорошим решением.

Единственным способом предотвращения этого является отключение двоичного журнала и установка параметра `innodb_support_xa` в 0[1].

Такие установки параметров небезопасны и несовместимы с репликацией. Для репликации необходимы и двоичный журнал, и поддержка XA, а кроме того — чтобы уже все было безопасно — параметр need `sync_binlog` должен быть равен 1, тогда подсистема хранения и двоичный журнал будут синхронизированы (в противном случае поддержка XA бессмысленна, так как двоичный журнал не может «закоммититься» на диске). Это одна из причин, по которым мы настоятельно рекомендуем использовать RAID-контроллер, оборудованный кэшем записи с аварийным электропитанием: кэш может ускорить выполнение дополнительных вызовов `fsync()` и нормализовать производительность.

В следующей главе мы подробно рассмотрим, как конфигурировать журнал транзакций и двоичный журнал.

Внешние XA-транзакции

MySQL может быть участником распределенной транзакции, но не может ею управлять. Она не поддерживает спецификацию XA в полном объеме. Например, спецификация XA позволяет соединять в одной транзакции таблицы из разных баз данных, но в настоящее время в MySQL это невозможно.

Внешние XA-транзакции еще более затратны, чем внутренние, из-за дополнительных задержек и большей вероятности того, что один из участников допустит ошибку. Использовать XA в сети WAN, а тем более в Интернете не рекомендуется из-за непредсказуемого поведения сети. Целесообразно избегать XA-транзакций, когда имеется хотя бы один непредсказуемый компонент, например медленная сеть или пользователь, который долго не нажимает кнопку Сохранить. Все, что способно задержать коммит, очень затратно, поскольку может вызвать задержку не в одной, а во многих системах.

Однако есть другие способы организации высокопроизводительных распределенных транзакций. Например, можно вставить данные локально и поставить обновление в очередь, а затем атомарно распространить его в составе гораздо более маленькой, быстрой транзакции. Можно также использовать репликацию MySQL для доставки данных из одного места в другое. Мы обнаружили, что некоторые приложения, использующие распределенные транзакции, вообще не нуждаются в них.

Несмотря на все сказанное, XA-транзакции могут оказаться полезным способом синхронизации данных между серверами. Этот метод хорошо работает, когда по какой-то причине применять репликацию невозможно или скорость выполнения обновлений некритична.

---

[1] Существует распространенное заблуждение, что innodb_support_xa необходим, только если вы используете транзакции XA. Это неверно: он контролирует внутренние транзакции XA между подсистемой хранения и двоичным журналом, и если вы цените свои данные, этот параметр должен быть включен.

Кэш запросов MySQL

Многие СУБД умеют кэшировать планы выполнения запросов, что позволяет серверу пропустить стадии разбора и оптимизации для встречавшихся ранее запросов. MySQL при некоторых условиях тоже может это делать, но, кроме того, у нее имеется кэш особого вида — так называемый *кэш запросов*, в котором хранятся *полные результирующие наборы*, сгенерированные командами SELECT. Этот раздел посвящен именно такому кэшу.

В кэше запросов MySQL хранятся именно те биты, которые запрос вернул клиенту. При попадании в кэш запросов сервер сразу же возвращает сохраненные итоги, пропуская стадии разбора, оптимизации и выполнения.

Кэш запросов отслеживает, какие таблицы были использованы в запросе, и если хотя бы одна из них изменилась, данные в кэше становятся недействительными. Такая грубая политика определения недействительности может показаться неэффективной, поскольку внесенные изменения, возможно, не отражаются на сохраненных результатах. Однако это простой подход с небольшими накладными расходами, что очень важно для сильно нагруженной системы.

Кэш запросов спроектирован так, что остается абсолютно прозрачным для приложений. Программе не нужно знать, получила она данные из кэша или в результате реального выполнения запроса. В любом случае результат будет одним и тем же. Другими словами, кэш запросов не изменяет семантику: с точки зрения приложения поведение сервера не зависит от того, активизирован кэш или нет[1].

Когда серверы стали более крупными и мощными, выяснилось, что кэш запросов, к сожалению, не был масштабируемой частью MySQL. Это фактически единственная точка конкуренции для всего сервера, поэтому она может вызвать серьезные стопоры на многоядерных серверах. Мы подробно рассмотрим, как его правильно настроить, однако считаем, что лучший подход — это сделать его по умолчанию отключенным и настроить небольшой кэш запросов (не более нескольких десятков мегабайт), но только если это будет полезно. В дальнейшем мы объясним, как определить, будет ли кэш запросов полезным для рабочей нагрузки.

Как MySQL проверяет попадание в кэш

Для проверки попадания в кэш MySQL применяет простую и очень быструю методику: по сути, кэш — это справочная таблица. Ключом этой таблицы является хеш, рассчитанный по тексту запроса, текущей базе данных, номеру версии клиентского протокола и еще нескольким параметрам, от которых могут зависеть результаты обработки запроса.

---

[1] На самом деле кэш запросов все-таки изменяет семантику в одном весьма тонком отношении: по умолчанию запрос может быть обслужен из кэша, если одна из участвующих в нем таблиц заблокирована предложением LOCK TABLES. Этот режим можно отменить, установив переменную query_cache_ wlock_invalidate.

При проверке попадания в кэш MySQL не разбирает, не нормализует и не параметризует команду — текст команды и прочие данные используются в том же виде, в каком получены от клиента. Любое различие, будь то регистр символов, лишние пробелы или комментарии, приведет к тому, что новый запрос не совпадет с кэшированной версией[1]. Об этом стоит помнить при написании запросов. Следовать единому стилю или способу форматирования — вообще хорошая привычка, но в данном случае она может еще и ускорить работу системы.

Следует также иметь в виду, что результат не попадет в кэш запросов, если сгенерирован недетерминированным запросом. Иначе говоря, любой запрос, содержащий недетерминированную функцию, например `NOW()` или `CURRENT_DATE()`, не кэшируется. Аналогично такие функции, как `CURRENT_USER()` и `CONNECTION_ID()`, дают разные результаты в зависимости от того, каким пользователем вызваны, поэтому также препятствуют записи запроса в кэш. Фактически кэш не работает для запросов, в которых есть ссылки на функции, определенные пользователем, хранимые процедуры, пользовательские переменные, временные таблицы, таблицы в базе данных `mysql` и таблицы, для которых заданы привилегии на уровне столбцов (полный перечень всех причин, по которым запрос не кэшируется, вы найдете в руководстве по MySQL).

Нам часто доводилось слышать, будто MySQL не проверяет кэш, если запрос содержит недетерминированную функцию. Это неверно. MySQL не может знать, детерминирована функция или нет, пока не разберет запрос, а поиск в кэше выполняется *до* разбора. Сервер лишь проверяет, что запрос начинается с букв `SEL` (без учета регистра), этим и ограничивается.

Однако сказать, что сервер не найдет в кэше результатов, если запрос содержит функцию типа `NOW()`, будет правильно, поскольку, даже если бы сервер и выполнял такой запрос раньше, он не поместил бы его в кэш. MySQL помечает запрос как некэшируемый, как только обнаруживает конструкцию, препятствующую кэшированию, и результаты такого запроса не сохраняются.

Есть полезный прием, позволяющий все же закэшировать запросы, ссылающиеся на текущую дату. Для этого нужно включить дату в виде литерала вместо использования функции, например:

```
... DATE_SUB(CURRENT_DATE, INTERVAL 1 DAY) - Не кэшируется!
... DATE_SUB('2007-07-14', INTERVAL 1 DAY) - Кэшируется
```

Поскольку кэш запросов работает с полным текстом команды `SELECT`, идентичные запросы, сделанные внутри подзапроса или представления, равно как и запросы внутри хранимых процедур, не могут воспользоваться кэшем. В версиях до MySQL 5.1 подготовленные запросы также не могли быть закэшированы.

---

[1] Percona Server является исключением из этого правила: он может отделять комментарии от запросов перед их сопоставлением с кэшем запросов. Эта функция необходима, поскольку вставка комментариев в запросы с дополнительной информацией о процессе, который их вызывал, — это обычная и хорошая практика. Ее, например, использует программное обеспечение для оснащения инструментами PHP, о котором мы говорили в главе 3.

Кэш запросов MySQL может повысить производительность, но, работая с ним, нужно помнить о некоторых тонкостях. Во-первых, включение кэша добавляет накладные расходы как при чтении, так и при записи:

- перед началом обработки запроса на чтение нужно проверить, есть ли он в кэше;
- если запрос допускает кэширование, но еще не помещен в кэш, то нужно потратить некоторое время на запись в кэш сгенерированных результатов;
- при обработке любого запроса на запись необходимо сделать недействительными все записи в кэше, в которых встречается измененная таблица. Этот процесс может быть очень ресурсозатратным, если кэш фрагментированный и/или очень большой (в нем хранится много кэшированных запросов или он настроен на использование большого объема памяти).

Кэш запросов все же может принести пользу. Однако, как мы увидим позже, дополнительные накладные расходы могут суммироваться, особенно в комбинации с параллелизмом, вызванным запросами, пытающимися заблокировать кэш для выполнения каких-либо действий с ним.

Во-вторых, пользователей InnoDB подстерегает еще одна проблема: полезность кэша ограничена транзакциями. Если какая-то команда внутри транзакции изменяет таблицу, то сервер делает недействительными все кэшированные запросы, ссылающиеся на нее, даже если реализованная в InnoDB система многоверсионного контроля способна скрыть в транзакции изменения, вызванные применением других команд. Таблица также считается глобально некэшируемой до коммита транзакции, поэтому все последующие запросы к ней — неважно, внутри или вне транзакции — также не могут кэшироваться, пока транзакция не завершится. Следовательно, длинные транзакции могут увеличить количество случаев непопадания в кэш.

Инвалидация может стать серьезной проблемой, когда кэш запросов велик. Если в нем много запросов, то поиск тех результатов, которые нужно объявить недействительными, может занять длительное время, в течение которого вся система будет простаивать. Так происходит потому, что существует единственная глобальная блокировка, разрешающая доступ к кэшу только одному запросу в каждый момент времени. А доступ выполняется как при проверке попадания, так и при выяснении того, какие запросы нужно сделать недействительными. В главе 3 приведен кейс, демонстрирующий накладные расходы на инвалидацию кэша.

Как кэш использует память

MySQL хранит весь кэш запросов в памяти, поэтому нужно понимать, как эта память используется, чтобы правильно настроить механизм кэширования. В кэше содержатся не только результаты запроса. В некоторых отношениях он очень похож на файловую систему: хранит структуры, позволяющие узнать, сколько памяти в пуле свободно, можно ли сопоставить таблицы и тексты команд с результатами запросов, текст запроса и его результат.

Если не принимать во внимание некоторые служебные структуры, под которые отведено примерно 40 Кбайт, то весь пул памяти, выделенной под кэш, состоит из *блоков* переменной длины. Каждый блок знает о своем типе, размере и о том, сколько данных он содержит, а также включает в себя указатели на следующий и предыдущий логический и физический блоки. Блоки могут быть нескольких типов: для хранения кэшированных результатов, списков таблиц, используемых в запросе, текста запроса и т. д. Однако блоки разных типов трактуются в основном одинаково, поэтому различать их для настройки кэша запросов нет необходимости.

На этапе запуска сервер инициализирует память, выделенную для кэша запросов. Первоначально пул памяти состоит из одного свободного блока. Его размер совпадает с размером сконфигурированной для нужд кэша памяти за вычетом служебных структур.

Когда сервер кэширует результаты запроса, он выделяет блок из пула памяти для хранения этих результатов. Минимальный размер блока составляет `query_ cache_ min_res_unit` байт, хотя может быть и бóльшим, если сервер знает, что для результирующего набора требуется больше памяти. К сожалению, сервер не в состоянии создать блок в точности подходящего размера, так как выделяет его еще до того, как результирующий набор полностью сгенерирован. Сервер не выстраивает весь набор в памяти, чтобы позже отправить его целиком, — гораздо эффективнее пересылать клиенту строки по мере их создания. Следовательно, начиная построение результирующего набора, сервер еще не знает, насколько большим он получится.

Выделение блока — довольно медленный процесс, так как сервер должен просмотреть список свободных блоков и найти среди них достаточно большой. Поэтому сервер пытается минимизировать количество операций выделения. Когда возникает необходимость кэшировать результирующий набор, сервер выделяет блок, размер которого не меньше минимально допустимого (он может быть больше — причины этого слишком сложны для объяснения), и начинает помещать в него результаты. Если блок заполняется раньше, чем данные закончились, то сервер выделяет новый блок — снова минимального или большего размера — и продолжает помещать в него данные. Когда результат полностью сгенерирован, сервер смотрит, осталось ли в последнем блоке место, и, если да, усекает блок и объединяет оставшуюся область с соседним свободным блоком. Весь этот процесс показан на рис. 7.3[1].

Говоря, что сервер выделяет блок, мы не имеем в виду, что он запрашивает память у операционной системы, вызывая функцию `malloc()` или ей подобную. Это делается только один раз, при создании кэша запросов. Затем сервер лишь просматривает список блоков и либо находит место, где собирается разместить новый блок, либо при необходимости удаляет самый старый из кэшированных запросов, чтобы освободить место. Другими словами, сервер MySQL самостоятельно управляет своей памятью, не полагаясь на операционную систему.

---

[1] Мы упростили рисунки в этом разделе, чтобы проиллюстрировать сказанное. На самом деле алгоритм выделения блоков кэша сложнее, чем здесь показано. Если вас интересуют детали, прочитайте комментарии в начале исходного файла sql/sql_cache.cc, там все очень понятно описано.

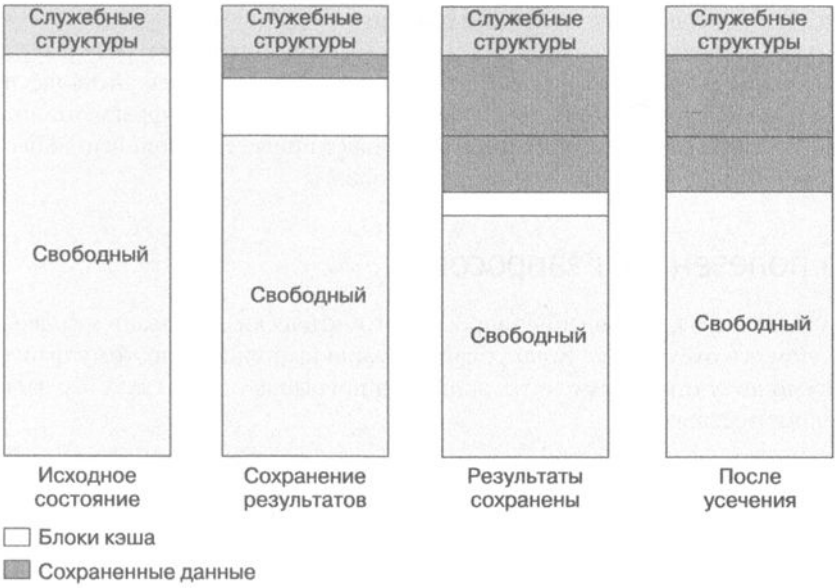


Рис. 7.3. Кэш запроса выделяет блоки для хранения результатов

До сих пор все выглядело довольно просто. Однако картина может оказаться сложнее, чем представлено на рис. 7.3. Предположим, что средний результирующий набор очень мал и сервер одновременно отправляет результаты по двум клиентским соединениям. После усечения может остаться блок, который меньше `query_cache_min_res_unit` и, следовательно, не может в будущем использоваться для хранения результирующих наборов. Выделение блоков может закончиться ситуацией, показанной на рис. 7.4.

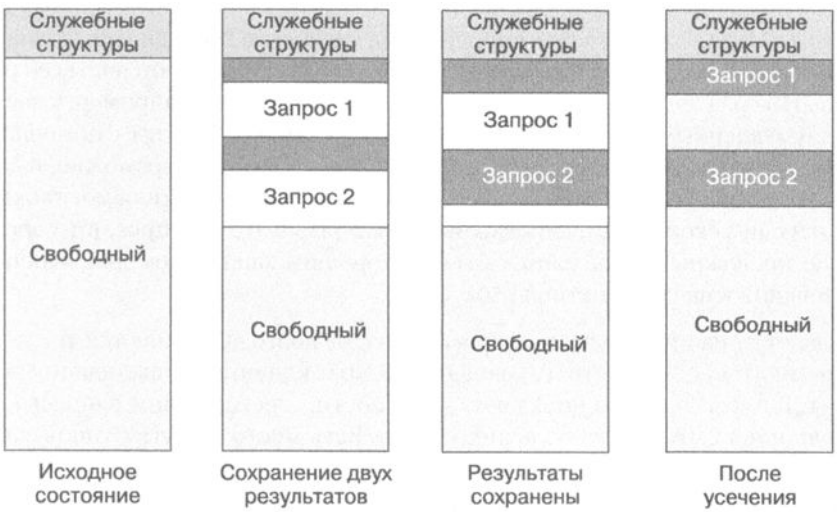


Рис. 7.4. Фрагментация, вызванная сохранением результатов в кэше запроса

После усечения первого результата по размеру между двумя результатами остается зазор — блок, слишком маленький для того, чтобы сохранить в нем результат другого запроса. Образование таких зазоров называется *фрагментацией*, это классическая проблема в задачах выделения памяти и в файловых системах. У фрагментации много причин, в том числе инвалидация кэша, она может привести к появлению настолько мелких блоков, что их нельзя больше использовать.

Когда полезен кэш запросов

Нельзя сказать, что кэширование запросов автоматически оказывается более эффективным, чем его отсутствие. Кэширование нужно выполнить, поэтому выигрыш от кэша можно получить, только если экономия превышает затраты. А это зависит от загруженности сервера.

Теоретически определить, полезен кэш или нет, можно, сравнив объем работы, которую сервер должен проделать с включенным и отключенным кэшем. Если кэш отключен, то любой запрос на чтение и запись должен быть выполнен, при этом чтение должно вернуть результирующий набор. Если кэш включен, то при обработке любого запроса на чтение необходимо сначала проверить кэш, а затем либо вернуть сохраненный результат, либо, если его нет в кэше, выполнить запрос, сгенерировать результирующий набор и отослать его клиенту. При обработке любого запроса на запись его следует сначала выполнить, а потом проверить, нужно ли провести инвалидацию каких-либо записей в кэше.

Выглядит все это довольно просто, однако подсчитать или предсказать выигрыш от использования кэша затруднительно. Необходимо принимать во внимание еще и внешние факторы. Например, кэш запросов может сократить время, необходимое для формирования результата, но не время, требующееся для отправки его клиенту, а именно оно может оказаться доминирующим фактором.

Кроме того, MySQL не позволяет определить, насколько полезен кэш запросов для отдельных запросов[1], поскольку счетчики в `SHOW STATUS` агрегируются по всей рабочей нагрузке. Но среднее поведение обычно не очень интересно. Например, у вас может быть один медленный запрос, который становится намного быстрее с помощью кэша запросов, но при этом он замедляет все остальное процессы, а возможно, и сам сервер. Это то, чего вы хотели добиться? На самом деле такие действия могут оказаться верными, если ускоряется очень важный для пользователя запрос, в то время как другие не так важны. Такой запрос был бы хорошим кандидатом для выборочного использования кэша с директивой `SQL_CACHE`.

Выигрывают от наличия кэша те запросы, которые долго выполняются, но занимают в кэше немного места, так что их хранение, возврат клиенту и инвалидация не очень затратны. В эту категорию попадают, в частности, агрегирующие запросы, например с функцией `COUNT()` для больших таблиц. Есть много и других типов запросов,

[1] Усовершенствованный «журнал медленных запросов» в Percona Server и MariaDB показывает, использовался ли для конкретных запросов кэш.

которые имеет смысл кэшировать. Эмпирическое правило гласит, что кэш запросов следует использовать, если в рабочей нагрузке преобладают сложные запросы `SELECT` с многотабличными соединениями, с фразами `ORDER BY` и `LIMIT`, которые создают небольшие результирующие наборы. У вас должно быть очень мало запросов `UPDATE`, `DELETE` и `INSERT` по сравнению с этими сложными запросами `SELECT`.

Один из самых простых способов понять, дает ли кэш выигрыш, — посмотреть на коэффициент попаданий в кэш. Он показывает, сколько запросов было обслужено из кэша, а не выполнено сервером. Когда сервер получает команду `SELECT`, он увеличивает одну из двух переменных состояния, `Qcache_hits` или `Com_select`, в зависимости от того, был запрос кэширован или нет. Поэтому коэффициент попаданий в кэш вычисляется по формуле `Qcache_hits / (Qcache_hits+Com_select)`.

К сожалению, коэффициент попаданий в кэш сложно интерпретировать. Какой коэффициент считать хорошим? Это зависит от разных факторов. Даже 30 % попаданий может быть очень неплохим результатом, поскольку экономия от невыполненных запросов, как правило (в расчете на один запрос), значительно превышает накладные расходы на объявление записей недействительными и сохранение результатов в кэше. Важно также знать, какие именно запросы кэшируются. Если попадания в кэш относятся к наиболее затратным запросам, то даже низкий коэффициент может означать существенную экономию на работе сервера. Таким образом, простого ответа на этот вопрос не существует.

Любой запрос `SELECT`, не обслуженный из кэша, называется *непопаданием в кэш*. Не попасть в кэш можно по следующим причинам.

- Запрос не допускает кэширования, поскольку либо он содержит недетерминированную конструкцию (например, функцию `CURRENT_DATE`), либо результирующий набор слишком велик для сохранения в кэше. В обоих случаях переменная `Qcache_not_cached` увеличивается на единицу.
- Сервер еще не видел такой запрос, поэтому не мог закэшировать его результат.
- Результат запроса был ранее закэширован, но сервер удалил его. Такое может произойти из-за нехватки памяти, из-за того, что кто-то велел серверу удалить запрос из кэша, или потому, что запись была инвалидирована (скоро расскажем чуть больше о инвалидации).

Если количество непопаданий в кэш велико, а количество некэшируемых запросов очень мало, то верно одно из следующих утверждений.

- Кэш еще не «прогрелся», то есть сервер не успел заполнить его результатами запросов.
- Сервер постоянно видит запросы, которые раньше не встречались. Если количество повторяющихся запросов невелико, то такое может случиться, даже когда кэш «прогрелся».
- Записи в кэше слишком часто инвалидируются (объявляются недействительными).

Инвалидация кэша может произойти из-за фрагментации, нехватки ресурсов или изменения данных. Если вы отвели под кэш достаточно памяти и правильно настроили

параметр query_cache_min_res_unit, то инвалидация по большей части обусловлена изменением хранимых значений. Узнать, сколько запросов модифицировали данные, позволяют переменные состояния Com_* (Com_update, Com_delete и т. п.), а сколько запросов было инвалидировано из-за нехватки памяти — переменная Qcache_lowmem_prunes.

Имеет смысл рассматривать затраты на инвалидацию отдельно от коэффициента попаданий. Рассмотрим предельный случай. Предположим, что из одной таблицы производится только чтение, при этом коэффициент попадания равен 100 %, а в другой — только обновление. Если вычислить коэффициент попадания по переменным состояния, то он составит 100 %. Но даже в этом случае использование кэша может оказаться неэффективным, поскольку замедляет выполнение запросов на обновление. При обработке любого запроса на обновление приходится проверять, не нужно ли объявить недействительными какие-то кэшированные данные, но, поскольку ответ неизменно будет «нет», вся эта работа проделана впустую. Такого рода проблему можно и не заметить, если не проанализировать совместно количество некэшируемых запросов и коэффициент попадания.

Сервер, для которого смесь операций записи и кэшируемого чтения из одних и тех же таблиц сбалансирована, также может не выиграть от наличия кэша запросов. Любая операция записи инвалидирует кэш, тогда как любая операция чтения вставляет в кэш новый запрос. А выигрыш можно получить, только если эти запросы потом обслуживаются из кэша.

Если закэшированный результат становится недействительным до того, как сервер снова получает ту же самую команду SELECT, вся работа по сохранению оказывается пустой тратой времени и памяти. Чтобы понять, имеет ли место такая ситуация, проверьте значения переменных Com_select и Qcache_inserts. Если почти все команды SELECT не попадают в кэш (то есть увеличивается значение Com_select), после чего их результирующие наборы кэшируются, то значение Qcache_inserts будет мало отличаться от Com_select. А хотелось бы, чтобы значение Qcache_inserts было существенно меньше Com_select, по крайней мере после «прогрева» кэша. Однако коэффициент по-прежнему сложно интерпретировать из-за тонкостей того, что происходит внутри кэша и сервера.

Как видите, коэффициент попадания в кэш и соотношение insert и select не сильно помогают. На самом деле целесообразнее всего измерить и рассчитать, насколько кэш может улучшить вашу рабочую нагрузку. Но если хотите, можно посмотреть на другое соотношение — коэффициент попадания во вставку. Этот показатель отражает отношение Qcache_hits к Qcache_inserts. В качестве грубого эмпирического правила соотношение «попадание во вставку» 3:1 или выше можно рассматривать как приемлемое для запросов средней скорости, однако соотношение 10:1 или выше гораздо лучше. Если вы не достигли такого уровня выгоды от кэша запросов, скорее всего, стоит отключить его. Исключением может быть ситуация, когда вы провели расчеты и определили, что для вашего сервера верны следующие моменты: попадания в кэш намного менее затратны, чем промахи, а параллелизм для кэша запросов не является проблемой.

Для каждого приложения существует конечный *потенциальный размер кэша* даже при отсутствии операций записи. Потенциальный размер — это объем памяти, необходимый для сохранения всех кэшируемых запросов, которые может выполнить приложение. В теории для большинства случаев этот размер очень велик. Но на практике у многих приложений он гораздо меньше, чем можно было бы ожидать, из-за того что большинство записей в кэше инвалидируются. Даже если выделить для кэша запросов очень много памяти, он никогда не заполнится больше чем на потенциальный размер кэша.

Необходимо следить за тем, какая доля выделенной для кэша памяти реально используется. Если используется не вся память, уменьшите размер кэша, если из-за нехватки памяти часто происходит вытеснение — увеличьте его. Впрочем, как уже было сказано, может быть опасно превысить несколько десятков мегабайт (это зависит от вашего оборудования и рабочей нагрузки).

Необходимо также соразмерять размер кэша запросов с другими серверными кэшами, например буферным пулом InnoDB или кэшем ключей MyISAM. Невозможно предложить коэффициент или простую формулу на все случаи жизни, поскольку подходящий баланс зависит от конкретного приложения.

Лучший способ узнать, насколько полезен кэш запросов, — измерить, если возможно, сколько времени запросы выполняются с кэшем и без него. Расширенный журнал медленных запросов Percona Server может сообщить, обслуживался запрос из кэша или нет. Если кэш запросов не сэкономит вам значительное количество времени, стоит, по-видимому, попробовать отключить его.

Как настраивать и обслуживать кэш запросов

Если вы понимаете, как работает кэш запросов, то настроить его довольно легко. В нем всего несколько «движущихся деталей».

- `query_cache_type`. Включен ли режим кэширования запросов. Допустимые значения: `OFF`, `ON`, `DEMAND`. Последнее означает, что кэшируются только запросы с модификатором `SQL_CACHE`. Эта переменная может быть установлена глобально или на уровне сеанса. (Подробнее о глобальных и сеансовых переменных см. в главе 6.)
- `query_cache_size`. Общий объем памяти, отведенной под кэш запросов, в байтах. Значение должно быть кратно 1024, поэтому MySQL, возможно, слегка изменит заданное вами число.
- `query_cache_min_res_unit`. Минимальный размер выделяемого блока. Ранее мы рассказали об этом параметре, дополнительная информация будет представлена в следующем разделе.
- `query_cache_limit`. Размер максимального результирующего набора, который разрешено кэшировать. Если при выполнении запроса результирующий набор оказывается больше, он не кэшируется. Помните, что сервер кэширует результаты по мере их генерации, поэтому заранее неизвестно, поместится ли результат в кэш. Если размер результата превышает заданный порог, то MySQL увеличивает

переменную состояния `Qcache_not_cached` и отбрасывает закэшированные к этому моменту строки. Если это происходит часто, можно включить подсказку `SQL_NO_CACHE` в запросы, которые, по вашему мнению, не должны вызывать возникновение этих накладных расходов.

- `query_cache_wlock_invalidate`. Следует ли обслуживать из кэша результаты, которые относятся к таблицам, заблокированным другими соединениями. По умолчанию значение этого параметра `OFF`, что изменяет семантику запроса, поскольку позволяет серверу читать кэшированные данные из заблокированной таблицы, хотя обычно сделать это невозможно. Если установить параметр равным `ON`, то чтение данных будет запрещено, но может увеличиться время ожидания блокировки. Для большинства приложений это несущественно, поэтому значение, принимаемое по умолчанию, как правило, приемлемо.

В принципе, настройка кэша довольно проста, сложнее разобраться в эффекте внесенных изменений. В следующих разделах мы попытаемся помочь вам научиться принимать правильные решения.

Уменьшение фрагментации

Полностью избежать фрагментации невозможно, но тщательный выбор параметра `query_cache_min_res_unit` может помочь не расходовать понапрасну память, отведенную под кэш запросов. Хитрость в том, чтобы найти оптимальное соотношение размера каждого нового блока с количеством операций выделения памяти, которые сервер должен выполнить во время сохранения результатов. Если задать слишком маленькое значение, то сервер будет терять меньше памяти, но блоки ему придется выделять чаще, а значит, объем работы увеличится. Если же размер слишком велик, то возрастет фрагментация. Придется искать компромисс между напрасным расходованием памяти и затратами процессорного времени на выделение блоков.

Оптимальное значение зависит от размера результирующего набора типичного запроса. Среднюю величину кэшированных запросов можно найти, разделив объем используемой памяти (приблизительно равен `query_cache_size - Qcache_free_memory`) на значение переменной состояния `Qcache_queries_in_cache`. Если имеется смесь больших и маленьких результирующих наборов, то, возможно, не удастся подобрать размер так, чтобы одновременно избежать фрагментации и минимизировать выделение блоков. Однако у вас могут быть основания полагать, что кэширование больших результатов не дает заметного выигрыша (часто так оно и есть). Подавить кэширование больших результирующих наборов можно, уменьшив значение переменной `query_cache_limit`. Иногда это позволяет достичь лучшего баланса между фрагментацией и накладными расходами на сохранение результатов в кэше.

Обнаружить, что кэш фрагментирован, можно, проанализировав переменную состояния `Qcache_free_blocks`, которая показывает количество блоков типа FREE. В конечной конфигурации, изображенной на рис. 7.4, есть два свободных блока. Наихудший случай фрагментации — это когда между каждой парой блоков, в которых хранятся данные, находится блок, размер которого чуть меньше минимального, то есть каждый второй блок свободен. Поэтому, если переменная `Qcache_free_blocks` приблизитель-

но равна Qcache_total_blocks / 2, то кэш сильно фрагментирован. Если переменная состояния Qcache_lowmem_prunes увеличивается и имеется много свободных блоков, значит, из-за фрагментации запросы преждевременно вытесняются из кэша.

Дефрагментировать кэш запросов можно с помощью команды FLUSH QUERY CACHE. Она уплотняет кэш, перемещая все блоки наверх и избавляясь от свободного пространства между ними, так что в итоге остается единственный свободный блок внизу. Несмотря на название, команда FLUSH QUERY CACHE не удаляет запросы из кэша. Для этой цели предназначена команда RESET QUERY CACHE. На время выполнения команды доступ к кэшу запросов блокирован, то есть, по существу, работа всего сервера приостановлена, так что будьте внимательны. Эмпирическое правило, относящееся к размеру кэша запросов, гласит: делайте его довольно маленьким, чтобы остановки, вызванные командой FLUSH QUERY CACHE, были короткими.

Улучшение использования кэша запросов

Если кэш не фрагментирован, но коэффициент попадания все равно низкий, то, возможно, вы отвели под него слишком мало памяти. Если сервер не может найти свободный блок, достаточно большой для сохранения нового результата, он должен вытеснить из кэша какие-то запросы.

При вытеснении запроса сервер увеличивает переменную состояния Qcache_lowmem_prunes. Если ее значение быстро увеличивается, это может быть вызвано двумя причинами.

- Если свободных блоков много, то виновата, скорее всего, фрагментация (см. предыдущий раздел).
- Если свободных блоков мало, возможно, рабочая нагрузка рассчитана на кэш большего размера, чем выделенный. Узнать, сколько в кэше неиспользуемой памяти, можно с помощью переменной Qcache_free_memory.

Если свободных блоков много, фрагментация низкая, вытеснений из-за нехватки памяти мало, а коэффициент попадания все еще невысок, то, вероятно, кэш запросов вообще мало помогает при данной загрузке сервера. Что-то мешает его использовать. Возможно, причина в большом количестве обновлений или в том, что запросы не допускают кэширования.

Если вы определили коэффициент попадания в кэш, но все же не уверены, выигрывает ли сервер от наличия кэша, можете отключить его, последить за производительностью, затем снова включить и посмотреть, как производительность изменится. Чтобы отключить кэш запросов, установите параметр query_cache_size в 0 (изменение параметра query_cache_size глобально не отражается на уже открытых соединениях и не приводит к возврату памяти серверу). Можно также заняться эталонным тестированием, но иногда бывает сложно получить реалистичную комбинацию кэшированных запросов, некэшированных запросов и обновлений.

На рис. 7.5 изображена блок-схема простой процедуры анализа и настройки кэша запросов.

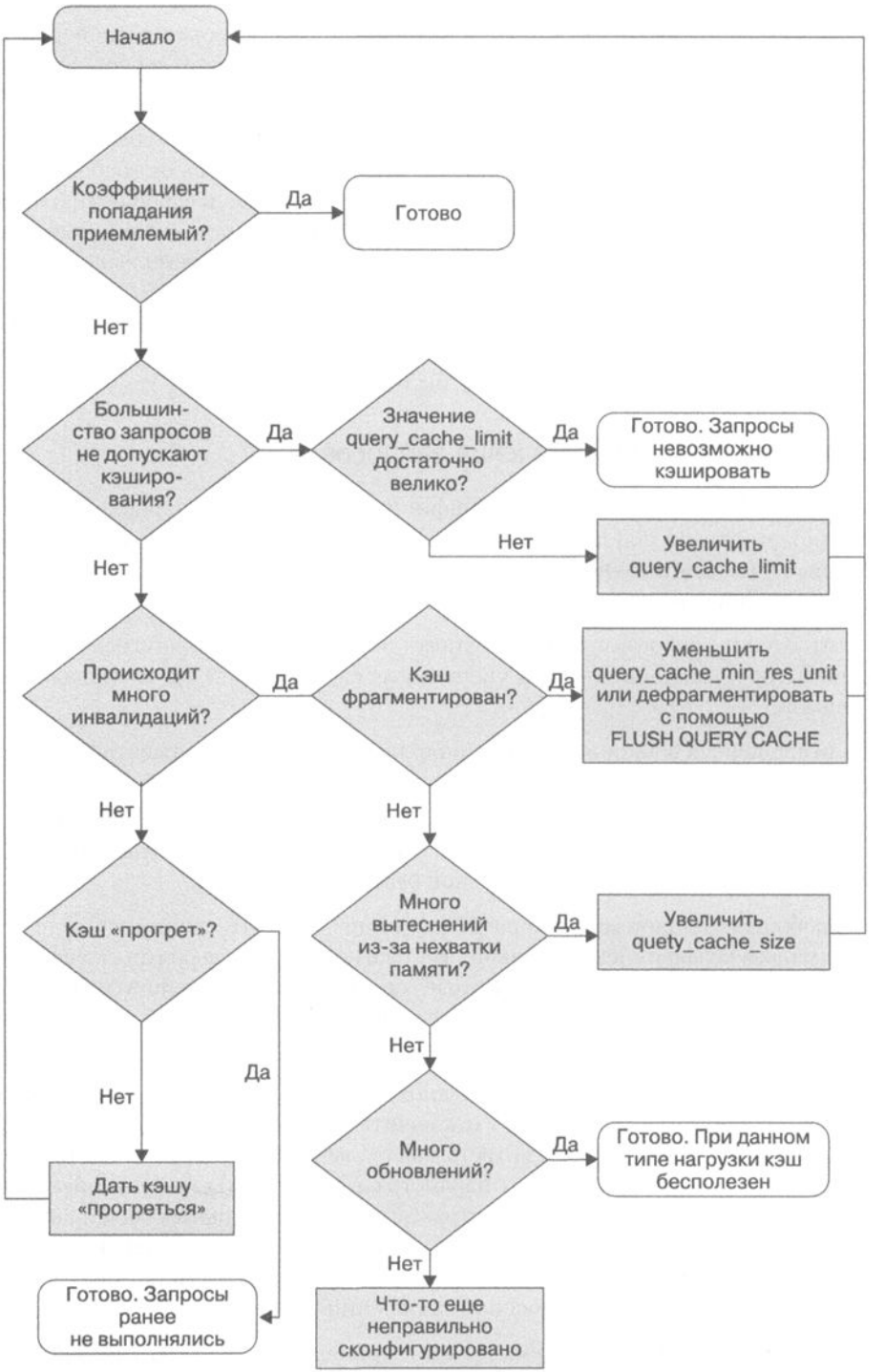


Рис. 7.5. Анализ и настройка кэша запросов

InnoDB и кэш запросов

InnoDB взаимодействует с кэшем более сложным образом, чем другие подсистемы хранения, поскольку она реализует MVCC. В MySQL 4.0 кэш запросов внутри транзакций вообще отключен, но в версии 4.1 и более поздних InnoDB сообщает серверу (для каждой таблицы в отдельности), может ли транзакция обращаться к кэшу запросов. Она управляет доступом к кэшу как для операций чтения (выборки результатов из кэша), так и для операций записи (сохранения результатов в кэше).

Возможность доступа определяется идентификатором транзакции и наличием блокировок на таблицу. В словаре данных InnoDB, который хранится в памяти, с каждой таблицей ассоциирован счетчик идентификаторов транзакций. Тем транзакциям, идентификатор которых меньше этого счетчика, запрещено выполнять операции чтения из кэша и записи в кэш для запросов, в которых участвует данная таблица.

Наличие любых блокировок на таблицу также препятствует кэшированию обращающихся к ней запросов. Например, если в транзакции выполняется запрос `SELECT FOR UPDATE` для некоторой таблицы, то никакая другая транзакция не сможет ни читать из кэша, ни писать в кэш запросы к этой же таблице до тех пор, пока все блокировки не будут сняты.

После выполнения коммита транзакции InnoDB обновляет счетчики для таблиц, на которые во время транзакции были поставлены блокировки. Наличие блокировки можно рассматривать как грубый эвристический подход для определения того, была ли таблица модифицирована внутри транзакции. Вполне возможно, что транзакция поставила блокировки на строки таблицы, но не обновила их. Однако не может быть так, чтобы данные в таблице были изменены без установки блокировок. InnoDB записывает в счетчик каждой таблицы максимальный системный идентификатор транзакции из существующих на текущий момент.

Отсюда вытекает следующее.

- Счетчик таблицы — это абсолютная нижняя граница транзакций, которым разрешено использовать кэш запросов. Если системный идентификатор транзакции равен 5 и транзакция установила блокировки на строки в таблице, а затем была закоммичена, то транзакции с первой по четвертую никогда не смогут обратиться к кэшу запросов для чтения или записи, если в запросе участвует эта таблица.
- В счетчик таблицы записывается не идентификатор транзакции, которая заблокировала строки в ней, а системный идентификатор транзакции. В результате транзакции, которые блокируют строки в некоторой таблице, могут оказаться не в состоянии читать из кэша или писать в него для будущих запросов, в которых участвует эта таблица.

Запись в кэш, выборка из него и инвалидация — все это выполняется на уровне сервера, поэтому InnoDB не в состоянии ни обойти эти механизмы, ни отложить операции. Однако подсистема хранения может явно попросить сервер инвалидировать запросы, в которых участвуют определенные таблицы. Это необходимо, когда

ограничение внешнего ключа, например `ON DELETE CASCADE`, изменяет содержимое таблицы, не упомянутой в запросе.

В принципе, архитектура MVCC в InnoDB допускает обслуживание запросов из кэша в случае, когда изменения в таблице не затрагивают согласованное представление, видимое другим транзакциям. Однако реализовать это было бы сложно. Для простоты алгоритмы InnoDB срезают некоторые углы ценой запрета доступа к кэшу запросов из транзакций даже в тех случаях, когда без этого можно было бы обойтись.

Общие оптимизации кэша запросов

Многие решения, касающиеся проектирования схемы, запросов и приложения, так или иначе затрагивают кэш запросов. В дополнение к тому, о чем было рассказано в предыдущих разделах, мы хотели бы отметить еще несколько моментов.

- Наличие нескольких маленьких таблиц вместо одной большой часто повышает эффективность использования кэша запросов. Таким образом, стратегия инвалидации работает на более мелком уровне. Но не придавайте этому соображению решающего значения при проектировании схемы, поскольку другие факторы могут легко перевесить все получаемые таким образом выгоды.
- Более эффективно собирать операции записи в пакет, чем выполнять их поодиночке, поскольку при таком подходе инвалидация запросов в кэше выполняется всего один раз.
- Мы обратили внимание на то, что сервер может на длительное время зависать, когда занимается инвалидацией записей или вытеснением запросов из большого кэша. Возможное решение — не задавать слишком большое значение параметра `query_cache_size`, но в некоторых случаях вы можете его просто отключить, поскольку нет предела совершенству.
- Невозможно контролировать кэш запросов на уровне отдельной базы данных или таблицы, но можно включать или выключать кэширование отдельных запросов с помощью модификаторов `SQL_CACHE` или `SQL_NO_CACHE` в команде `SELECT`. Можно также включать или выключать кэширование для отдельного соединения, установив подходящее значение сеансовой переменной `query_cache_type`.
- Для приложений, выполняющих много операций записи, полное отключение кэширования может повысить производительность. При этом исключаются накладные расходы на кэширование запросов, которые в скором времени все равно будут инвалидированы. Напомним, что кэширование отключается установкой параметра `query_cache_size` равным `0`, так что кэш при этом не потребляет памяти.
- Отключение кэша запросов может оказаться полезным также для приложений, выполняющих много операций чтения, из-за конкуренции за мьютекс кэша запроса. Если вам нужна хорошая производительность при высоком уровне параллелизма, обязательно проверяйте ее с помощью тестов в соответствующих условиях, поскольку включение кэша запросов и тестирование при низком параллелизме могут быть очень обманчивыми.

Если вы хотите избежать кэширования большинства запросов, но знаете, что некоторые из них смогли бы сильно выиграть от кэширования, то присвойте глобальному параметру `query_cache_type` значение `DEMAND`, а затем включите в нужные запросы подсказку `SQL_CACHE`. Это более трудоемко, зато дает более точный контроль над кэшем. Наоборот, если требуется кэшировать большинство запросов, а исключить лишь некоторые, то можно включить в них указание `SQL_NO_CACHE`.

Альтернативы кэшу запросов

В основу работы кэша запросов MySQL положен простой принцип: быстрее всего обрабатывается запрос, который вообще не нужно выполнять. Однако все равно приходится отправлять запрос, а серверу — что-то с ним делать, пусть и не очень много. Но что если вам вообще не придется обращаться к серверу с конкретным запросом? Кэширование на стороне клиента может еще больше снизить нагрузку на сервер. Мы вернемся к этому вопросу в главе 14.

Итоги главы

Эта глава, в отличие от предыдущих, была своеобразным попурри из разных тем. Подведем итоги, кратко перечислив важные моменты каждой темы.

- *Секционирование таблиц.* Секционирование — это своего рода дешевое, грубое индексирование, которое работает для больших объемов. Для достижения наилучших результатов либо забудьте об индексировании и планируйте полное секционирование, либо убедитесь, что только к одной секции выполняется большинство запросов и она помещается в памяти вместе со своими индексами.

  Делайте до 150 секций на таблицу, отслеживайте затраты для каждой строки и для каждого запроса при секционировании.
- *Представления.* Представления могут быть полезны для абстрагирования базовых таблиц и сложных запросов. Однако остерегайтесь представлений, которые используют временные таблицы, поскольку они не опускают `WHERE` к базовым запросам. Также у них отсутствуют индексы, поэтому вы не сможете эффективно обращаться к ним при соединении. Самый лучший вариант — использовать представления просто ради удобства.
- *Внешние ключи.* Ограничения внешнего ключа приводят к тому, что ограничения применяются более эффективно — на сервере. Однако они могут увеличить сложность, вызвать появление дополнительных издержек на индексацию и взаимодействие между таблицами, что приведет к увеличению числа блокировок и повышению параллелизма. Мы считаем, что внешние ключи — это отличная возможность обеспечить целостность системы, но они являются предметом роскоши для приложений, требующих чрезвычайно высокой производительности. Большинство разработчиков, заботящихся о высокой производительности, не используют их, предпочитая доверять коду приложения.

- *Хранимые подпрограммы.* То, как в MySQL реализованы хранимые процедуры, триггеры, хранимые функции и события, откровенно говоря, не впечатляет. Существует также множество проблем с репликацией на основе команд. Используйте эти возможности тогда, когда они могут сэкономить много сетевых обращений, — в таких случаях вы сможете добиться лучшей производительности, сократив весьма серьезную задержку. Вы также можете использовать их по обычным причинам (централизация бизнес-логики, принудительное наделение привилегиями и т. д.). Однако эти подпрограммы работают в MySQL совсем не так, как на больших и более сложных серверах баз данных.
- *Подготовленные операторы.* Подготовленные операторы полезны, когда значительная часть затрат на выполнение команд заключается в передаче операторов по сети, разборе SQL-кода и его оптимизации. Если вы будете повторять один и тот же оператор много раз, то сможете сэкономить на этих затратах, используя подготовленные операторы, поскольку разбор кода выполняется только однажды, проводится кэширование плана выполнения, а бинарный протокол более эффективен, чем обычный текстовый протокол.
- *Плагины.* Плагины, написанные на языках C или C ++, позволяют расширить функциональность сервера разными способами. Они очень мощные. Мы написали много UDF и плагинов для различных целей, когда проблема лучше всего решается внутри сервера с помощью его естественного кода.
- *Кодировки.* Кодировка — это соответствие двоичного кода определенному набору символов. Схема упорядочения — это набор правил сортировки для конкретной кодировки. Большинство людей используют либо кодировку latin1 (кодировка по умолчанию, подходящая для английского и некоторых европейских языков), либо UTF-8. Если используете UTF-8, остерегайтесь временных таблиц и буферов: сервер выделяет 3 байта на символ, поэтому, если вы не будете осторожны, система будет потреблять много дискового пространства и памяти. Следите за тем, чтобы кодировки совпадали с параметрами настройки кодировок для всех клиентских соединений, в противном случае произойдет конвертация символов, что нарушит индексирование.
- *Полнотекстовый поиск.* На момент написания книги только MyISAM поддерживала полнотекстовые индексы. MyISAM в основном нельзя применять для широкомасштабного полнотекстового поиска из-за блокировки и отсутствия устойчивости к сбоям. Поэтому мы обычно рекомендуем использовать вместо этого Sphinx.
- *XA-транзакции.* Большинство не использует XA-транзакции с MySQL. Однако *не отключайте* `innodb_support_xa`, *если не знаете точно, что делаете.* Многие считают, что этот параметр необходим, только если XA-транзакции используются явно. Это *не так.* Он применяется для координации InnoDB и двоичного журнала, помогая восстановлению после сбоев.
- *Кэш запросов.* Кэш запросов позволяет не обрабатывать запросы повторно, если сохраненный результат идентичного запроса уже находится в кэше. Работая с кэшем запросов в системах с высокой нагрузкой, мы наблюдали множество случаев

блокировки серверов. Если вы применяете кэш запросов, не делайте его очень большим и используйте, только если уверены, что это принесет пользу. Как вы можете это узнать? Лучший способ — использовать расширенные службы журналирования Percona Server и выполнить несложные расчеты. Если вы не согласны с этим, можете посмотреть коэффициент попадания в кэш (не всегда полезно), соотношение `select` и `insert` (также трудно интерпретировать) или коэффициент попадания в вставку (немного более значимый). В общем, кэш запросов удобен, потому что прозрачен и не требует от вас дополнительного кодирования, но если нужен высокоэффективный кэш для получения высокой производительности, стоит подумать о memcached или другом внешнем решении. Подробнее об этом — в главе 14.

8 Оптимизация параметров сервера

Из этой главы вы узнаете, как создать хороший файл конфигурации для своего сервера MySQL. Ее можно сравнить с круизом, предусматривающим осмотр большого количества достопримечательностей и периодически отклоняющимся от маршрута для того, чтобы посмотреть живописные виды. Эти отклонения от прямого пути необходимы, поскольку поиск кратчайшего пути к хорошей конфигурации начинается не с изучения параметров конфигурации и определения того, какие из них вы должны установить или как изменить, и не с анализа поведения сервера и поиска ответа на вопрос, могут ли какие-то параметры конфигурации улучшить его. Лучше всего сначала разобраться во внутреннем устройстве и поведении MySQL. Затем эти знания стоит использовать в качестве руководства по настройке MySQL. Наконец, вы можете сравнить желаемую конфигурацию с текущей и внести изменения, которые будут важны и целесообразны именно для вас.

Нам часто задают вопросы примерно такого плана: «Как оптимально задать конфигурационные параметры для моего сервера с 32 Гбайт памяти и 12-ядерным процессором?» К сожалению, на этот вопрос нет однозначного ответа. Конфигурация сервера зависит от рабочей нагрузки, данных и используемых приложений, а не только от оборудования.

Конечно, вы можете изменить настройки MySQL, но не стоит этого делать. Как правило, лучше верно настроить базовые параметры (в большинстве случаев лишь несколько из них) и потратить больше времени на оптимизацию схемы, индексы и проектирование запросов. При правильной настройке базовых параметров конфигурации MySQL потенциальные выгоды от дальнейших изменений обычно малы.

В то же время возня с настройками может привести к огромным проблемам. Мы не раз видели «хорошо настроенные» серверы, которые постоянно ломались, зависали или медленно работали. Так что потратим немного времени на объяснение, почему что-то может произойти и чего делать не следует.

Итак, что нужно делать? Удостоверьтесь в том, что значения основных параметров, таких как размер буферного пула InnoDB и файла журнала, подходят для вашей системы. Далее, если хотите обеспечить хорошую работу, задайте несколько параметров, отвечающих за безопасность и исправность (но обратите внимание на то, что

они обычно не улучшают производительность — лишь помогают избежать проблем), а остальные настройки оставьте в покое. Если вы столкнетесь с проблемой, тщательно диагностируйте ее с помощью методов, описанных в главе 3. Если проблема вызвана компонентом сервера, поведение которого может быть исправлено с помощью параметра настройки, то измените его.

Кроме того, в некоторых случаях вам может потребоваться установить определенные параметры настройки, которые способны значительно повлиять на производительность. Однако учтите, что они не должны быть частью базового файла конфигурации сервера. Устанавливать их следует только в том случае, если вы столкнетесь с конкретными проблемами с производительностью, которые эти настройки могут решить. Именно поэтому мы советуем не искать специально параметры, которые можно улучшить, и не изменять их. Если какая-то проблема требует решения, она должна сначала проявиться во времени отклика на запрос. Лучше всего заниматься улучшениями, начав с запросов и времени отклика на них, а не с параметров настройки. Это поможет вам сэкономить время и предотвратить множество проблем.

Еще один хороший способ сэкономить время и усилия — использовать значения по умолчанию (если только вы точно не уверены в том, что их стоит изменить). Работа с настройками по умолчанию повышает безопасность системы, поскольку многие пользователи также сохраняют настройки по умолчанию. В итоге эти настройки можно считать наиболее тщательно проверенными. А при их изменении могут всплыть неожиданные ошибки.

Основы конфигурации MySQL

Сначала мы объясним работу механизма конфигурирования, а затем расскажем, что нужно настраивать в MySQL. Вообще говоря, MySQL снисходительно относится к ошибкам конфигурации, но наши замечания помогут сэкономить немало времени и сил.

Прежде всего запомните, откуда MySQL получает конфигурационную информацию: из аргументов командной строки и параметров в конфигурационном файле. В системах на базе UNIX таковым обычно является файл `/etc/my.cnf` или `/etc/mysql/my.cnf`. Если вы пользуетесь скриптами запуска, входящими в состав операционной системы, то обычно только в этом файле и надо определять конфигурацию. Если же запускаете MySQL вручную, например при тестовой установке, то можете задавать параметры и в командной строке. Фактически сервер считывает содержимое конфигурационного файла, удаляет из него строки комментариев и символы разрыва строки, а затем обрабатывает этот файл вместе с параметрами командной строки.

Замечания по терминологии: поскольку многие параметры командной строки MySQL соответствуют переменным сервера, мы иногда используем термины «параметр» и «переменная» как взаимозаменяемые. Большинство конфигурационных переменных называются так же, как соответствующие параметры командной строки, но есть несколько исключений. Например, параметр --memlock устанавливает переменную locked_in_memory.

Любые постоянные параметры следует помещать в глобальный конфигурационный файл, а не задавать в командной строке. В противном случае вы рискуете случайно запустить сервер без них. Кроме того, разумно хранить все конфигурационные файлы в одном месте, чтобы их проще было инспектировать.

Не забывайте, где находится конфигурационный файл вашего сервера! Нам доводилось встречать людей, которые безуспешно пытались настроить сервер, изменяя файл, который он и не собирался читать, например `/etc/my.cnf` в системе Debian GNU/Linux, где сервер ищет файл `/etc/mysql/my.cnf`. Иногда такие файлы могут находиться в разных местах, быть может, потому, что предыдущий системный администратор тоже запутался. Если вы не знаете, какие файлы читает сервер, то лучше у него же и спросить:

```
$ which mysqld
/usr/sbin/mysqld
$ /usr/sbin/mysqld --verbose --help | grep -A 1 'Default options'
Default options are read from the following files in the given order:
/etc/mysql/my.cnf ~/.my.cnf /usr/etc/my.cnf
```

Все это относится к типичным установкам для тех случаев, когда на машине только один сервер. Можно спроектировать и более сложные конфигурации, но все подобные способы нестандартны. Ранее в дистрибутив MySQL входила ныне устаревшая программа mysqlmanager, которая могла запускать несколько экземпляров сервера, используя один конфигурационный файл с несколькими разделами (она заменила еще более старый скрипт mysqld_multi). Однако во многих дистрибутивах операционных систем эта программа отсутствует или не используется в сценариях запуска. На самом деле операционные системы зачастую вообще игнорируют сценарии запуска, поставляемые с MySQL.

Конфигурационный файл разбит на разделы (секции), каждый из которых начинается со строки с именем секции в квадратных скобках. Любая программа, входящая в состав дистрибутива MySQL, обычно читает секцию, имя которой совпадает с именем самой программы. Кроме того, многие клиентские приложения читают секцию `client`, в которую можно поместить общие для всех клиентов параметры. Сервер обычно читает секцию `mysqld`. Следите за тем, чтобы параметры располагались в нужной секции, иначе эффект от них будет нулевой.

Синтаксис, область видимости и динамичность

Конфигурационные параметры записываются строчными буквами, слова разделяются символами подчеркиваниями или дефисами. Следующие формы записи эквивалентны, любую из них можно встретить в командной строке или конфигурационном файле:

```
/usr/sbin/mysqld --auto-increment-offset=5
/usr/sbin/mysqld --auto_increment_offset=5
```

Мы рекомендуем выбрать один из этих стилей и придерживаться его. Так будет проще искать в файле конкретный параметр.

У конфигурационной переменной может быть несколько областей видимости. Одни параметры действуют на уровне всего сервера (глобальная область видимости), другие могут задаваться по-своему для каждого соединения (сеансовая область видимости), третьи относятся к конкретным объектам. У многих сеансовых параметров есть глобальные эквиваленты, которые можно рассматривать как значения по умолчанию. Изменение сеансовой переменной отразится только на том соединении, где она была изменена, и после его закрытия изменения будут потеряны. Приведем несколько примеров разнообразного поведения, к которому следует быть готовыми.

- Переменная `query_cache_size` имеет глобальную область видимости.
- Переменная `sort_buffer_size` имеет глобальное значение по умолчанию, но может быть изменена на уровне сеанса.
- Переменная `join_buffer_size` имеет глобальное значение по умолчанию, может быть изменена на уровне сеанса, но, кроме того, для каждого запроса, в котором задействуется несколько таблиц, можно выделить по одному буферу *на операцию соединения*, то есть для одного запроса могут существовать несколько буферов соединения.

Переменные в конфигурационном файле не только можно устанавливать, но и изменять (хотя не все) во время работы сервера. Такие конфигурационные переменные в MySQL называются *динамическими*. Далее показаны разные способы динамического изменения значений переменной `sort_buffer_size` на уровне сеанса и глобально:

```
SET           sort_buffer_size  = <значение>;
SET GLOBAL    sort_buffer_size  = <значение>;
SET         @@sort_buffer_size := <значение>;
SET @@session.sort_buffer_size := <значение>;
SET  @@global.sort_buffer_size := <значение>;
```

Изменяя переменные динамически, не забывайте, что после останова MySQL новое значение исчезнет. Если хотите сохранить измененные параметры, то запишите их в конфигурационный файл.

Изменение глобальной переменной во время работы сервера не отражается на ее значении в текущем и во всех остальных открытых сеансах. Это связано с тем, что сеансовые переменные инициализируются в момент создания соединения. После любого изменения переменной выполняйте команду `SHOW GLOBAL VARIABLES`, чтобы удостовериться в том, что желаемый эффект достигнут.

Для разных переменных используются различные единицы измерения, и их следует знать. Например, переменная `table_cache` определяет количество кэшируемых таблиц, а не размер кэша таблиц в байтах. Переменная `key_buffer_size` задается в байтах, тогда как другие параметры могут измеряться и в других единицах, например процентах.

Для многих переменных можно указывать суффикс. Например, `1M` означает 1 Мбайт. Однако это работает только при задании переменной в конфигурационном файле или в аргументе командной строки. При использовании SQL-команды `SET` следует

указывать литеральное значение `1048576` или выражение, например `1024 * 1024`. В конфигурационных файлах выражения не допускаются.

В команде `SET` можно задавать также специальное значение `DEFAULT`. Если присвоить его сеансовой переменной, то она станет равна соответствующей глобальной переменной. Присваивание же значения `DEFAULT` глобальной переменной делает ее равной значению, заданному на этапе компиляции сервера (а не тому, которое указано в конфигурационном файле). Это бывает полезно, когда нужно вернуть переменной то состояние, которое она имела в момент открытия соединения. Мы не рекомендуем делать это по отношению к глобальным переменным, поскольку результат может отличаться от ожидаемого — вы не получите значение, действовавшее на момент запуска сервера.

Побочные эффекты установки переменных

Динамическое задание переменных может иметь неожиданные побочные эффекты, например сброс на диск изменившихся блоков из буферов. Будьте осторожны при онлайновом изменении переменных, так как это может сильно нагрузить сервер.

Иногда назначение переменной можно понять из ее имени. Например, переменная `max_heap_table_size` делает именно то, что подразумевает ее наименование: задает *максимальный* размер, до которого могут расти в памяти временные таблицы. Однако соглашение о наименованиях не всегда последовательно, поэтому догадаться о назначении переменной по названию можно далеко не всегда.

Рассмотрим некоторые важные переменные и последствия их динамического изменения.

- `key_buffer_size`. При задании этой переменной сразу же резервируется указанный объем памяти для буфера ключей (он также называется *кэшем ключей*). Однако операционная система не выделяет физическую память до момента ее фактического использования. Так, требование выделить 1 Гбайт под буфер ключей вовсе не означает, что сервер немедленно получит весь гигабайт (о том, как можно следить за использованием памяти сервером, расскажем в главе 9).

  Как объясняется далее в этой главе, MySQL позволяет создавать несколько кэшей ключей. Если присвоить этой переменной значение `0` для кэша ключей, отличного от подразумеваемого по умолчанию, то MySQL переместит все индексы из указанного кэша в кэш по умолчанию, а указанный кэш удалит, когда больше никто не будет его использовать. В результате задания этой переменной для несуществующего кэша указанный кэш будет создан. Присваивание данной переменной ненулевого значения для существующего кэша приводит к тому, что память, отведенная под него, сбрасывается на диск. Это действие блокирует все операции, пытающиеся получить доступ к кэшу, пока сброс не будет закончен.
- `table_cache`. Установка этой переменной не дает немедленного эффекта — действие откладывается до следующей попытки потока открыть таблицу. Когда это происходит, MySQL проверяет значение данного параметра. Если оно больше текущего количества таблиц в кэше, то поток может поместить вновь открытую таблицу в кэш. В противном случае MySQL удалит из кэша неиспользуемые таблицы.

- `thread_cache_size`. Установка этой переменной не дает немедленного эффекта — действие откладывается до момента следующего закрытия соединения. В этот момент MySQL проверит, есть ли в кэше место для хранения потока. Если да, то поток кэшируется для повторного использования в будущем другим соединением. Если нет — уничтожается. В этом случае количество потоков в кэше, а следовательно, и объем памяти, отведенной под кэш потоков, не сокращается сразу — уменьшение происходит только тогда, когда новое соединение позаимствует для себя поток из кэша (MySQL добавляет потоки в кэш только при закрытии соединения, а удаляет их из кэша лишь при создании новых соединений).
- `query_cache_size`. MySQL выделяет и инициализирует указанное количество памяти для кэша запросов в момент запуска сервера. При изменении этой переменной (даже если новое значение совпадает с предыдущим) MySQL немедленно удаляет все кэшированные запросы, устанавливает новый размер кэша и повторно инициализирует отведенную под него память. Этот процесс стопорит сервер и может занять довольно много времени, поскольку MySQL удаляет все кэшированные запросы один за другим, а не одновременно.
- `read_buffer_size`. MySQL не выделяет память для этого буфера, пока она не потребуется запросу. Когда же необходимость возникает, MySQL немедленно выделяет весь блок запрошенного размера.
- `read_rnd_buffer_size`. MySQL не выделяет память для этого буфера, пока она не потребуется запросу. Когда же необходимость возникает, MySQL выделяет лишь необходимый объем памяти (имя `max_read_rnd_buffer_size` точнее описывало бы назначение этой переменной).
- `sort_buffer_size`. MySQL не выделяет память для этого буфера, пока она не потребуется запросу, чтобы выполнить сортировку. Когда же необходимость возникнет, запрошенный блок немедленно выделяется целиком независимо от того, нужно столько памяти или нет.

В дальнейшем назначение этих переменных будет объяснено подробнее. И этот список не исчерпывающий. Пока мы лишь хотим показать, какого поведения ожидать при изменении этих важных параметров.

Не следует глобально увеличивать параметр, относящийся к соединению, если вы не уверены в правильности такого решения. Некоторые буферы выделяются одним куском, даже если они не нужны, поэтому глобальное изменение параметра может привести к растрачиванию памяти впустую. Лучше увеличивать значение, только когда это необходимо для конкретного запроса.

Типичный пример переменной, которую лучше оставить небольшой, а увеличивать только для некоторых запросов, — это `sort_buffer_size`, которая управляет размером буфера для файловой сортировки. MySQL выполняет некоторую работу по инициализации буфера сортировки после его выделения.

Кроме того, буфер сортировки выделяется целиком даже для очень маленьких сортировок, поэтому если вы сделаете его намного больше, чем требуется для средней сортировки, то впустую потратите память и увеличите затраты на ее распределение. Этот тезис может удивить тех читателей, которые считают распределение памяти не очень затратной операцией. Не углубляясь в технические детали, скажем, что

распределение памяти включает в себя настройку адресного пространства, которая может быть довольно утомительной. В частности, в Linux распределение памяти использует несколько стратегий с различными затратами в зависимости от размера.

Таким образом, большой буфер сортировки может быть очень затратным, поэтому не увеличивайте его размер, если не уверены, что это необходимо.

Обнаружив запрос, который мог бы выполняться быстрее при наличии большого буфера сортировки, вы можете увеличить значение `sort_buffer_size` непосредственно перед его выполнением для конкретной сессии, а затем вернуться к значению `DEFAULT`. Вот как это делается:

```
SET @@session.sort_buffer_size := <значение>;
-- Выполнить запрос...
SET @@session.sort_buffer_size := DEFAULT;
```

Для подобного кода могут пригодиться функции-обертки. К числу других переменных, которые разумно устанавливать на уровне соединения, относятся `read_buffer_size`, `read_rnd_buffer_size`, `tmp_table_size` и `myisam_sort_buffer_size` (для исправления таблиц).

Чтобы сохранить и восстановить значения переменной, заданные пользователем, можно написать примерно такой код:

```
SET @saveC_<имя_уникальной_переменной> := @@session.sort_buffer_size;
SET @@session.sort_buffer_size := <значение>;
-- Выполнить запрос...
SET @@session.sort_buffer_size := @saved_<имя_уникальной_переменной>;
```

Размер буфера сортировки — это один из параметров, который требует слишком тщательной настройки. Кажется, некоторые люди считают, что чем больше, тем лучше, и мы даже видели серверы, у которых значение этой переменной составляло 1 Гбайт. Скорее всего (что неудивительно), это приведет к тому, что сервер попытается выделить слишком много памяти и рухнет либо просто сожжет много процессорного времени при инициализации буфера сортировки для запроса (см. ошибку MySQL 37359 для получения дополнительной информации об этом).

Не придавайте слишком большого значения размеру буфера сортировки. Вам действительно нужно, чтобы запросы получили 128 Мбайт памяти для сортировки десяти строк и возврата их клиенту? Подумайте о том, какие виды сортировки выполняют запросы и как часто, и лучше постарайтесь их избежать, выполнив правильную индексацию и проектирование запросов (см. главы 5 и 6), а не пытайтесь быстрее провести операцию сортировки. Кроме того, следует обязательно выполнить профилирование своих запросов, чтобы посмотреть, будет ли сортировка тем процессом, на котором вам в любом случае придется сконцентрировать внимание (см. главу 3, где приведен пример запроса, который выполняет сортировку, но не тратит на нее много времени).

Приступая к работе

Будьте осторожны, устанавливая переменные. Больше — не всегда лучше: установив слишком большое значение, легко вызвать проблемы — памяти может не хватить, и тогда сервер начнет выгружать ее в файл подкачки или вообще выйдет за пределы адресного пространства[1].

Обязательно нужно иметь систему мониторинга, которая определяет, улучшилась или ухудшилась общая производительность сервера после изменения. Эталонного тестирования недостаточно, поскольку оно не является реальным использованием сервера. Если не измерять фактическую производительность, то можно навредить, даже не осознавая того. Мы не раз видели, как кто-то менял конфигурацию сервера, полагая, что повысил быстродействие, тогда как на деле общая производительность снижалась, поскольку в разное время дня или в разные дни недели нагрузка варьировалась.

Делая заметки, возможно, с комментариями в файле конфигурации, вы можете сэкономить себе и коллегам много времени и сил. Еще лучше поместить конфигурационный файл в систему управления версиями. Это хороший прием, потому что он позволяет вам откатывать изменения. Чтобы упростить обслуживание большого количества конфигурационных файлов, создайте символическую ссылку с конфигурационного файла на центральный репозиторий системы управления версиями.

Прежде чем приступать к изменению конфигурационных параметров, следует настроить запросы и схему, обращая внимание по крайней мере на такие очевидные оптимизации, как добавление индексов. Если вы слишком далеко продвинетесь по пути изменения конфигурации, а потом займетесь модификацией запросов или схемы, то настройку, возможно, придется начинать заново. Не забывайте, что настройка — это непрерывный итерационный процесс. Если только оборудование, рабочая нагрузка и данные не являются абсолютно неизменными, то вероятность того, что к конфигурированию придется вернуться, очень велика. На самом деле большинство серверов не имеют постоянной рабочей нагрузки даже в течение дня, а это означает, что идеальная конфигурация на утро не годится для полудня! Очевидно, что стремление создать мифическую идеальную конфигурацию совершенно нецелесообразно. Это означает, что не нужно пытаться выжать из сервера все до последней капли, ведь отдача от затраченного на это времени, скорее всего, будет крайне мала. Мы рекомендуем продолжать настройку до тех пор, пока не получится приемлемый результат, а потом ничего не трогать, пока не появится причина полагать, что можно добиться существенного повышения производительности.

---

[1] Мы часто сталкивались с такой ошибкой. При настройке сервера с вдвое большим объемом памяти, чем существующий сервер, в качестве ориентира используется файл конфигурации старого сервера. При этом все параметры конфигурации нового сервера получают умножением на два соответствующих параметров старого. Этот прием не работает.

Итеративная оптимизация с помощью эталонного тестирования

Возможно, вы полагаете, что можно сформировать эталонный тестовый набор и тщательно настроить ваш сервер, итеративно изменяя его конфигурацию в поисках оптимальных настроек. Обычно мы советуем этого не делать. Ведь потребуется много работы и исследований, а потенциальный выигрыш в большинстве случаев настолько мал, что, скорее всего, вы напрасно потратите время. Вероятно, лучше потратить его на другие задачи, например на проверку резервных копий, мониторинг изменений в планах запросов и т. п.

Кроме того, очень сложно понять, какие побочные эффекты могут быть вызваны внесенными изменениями в долгосрочной перспективе. Может случиться так, что вы измените параметр и результаты эталонного тестирования улучшатся. Однако эталонные тесты измеряют не все важные характеристики, либо вы можете запустить их на недостаточно длительный срок и они не обнаружат изменений в долговременном устойчивом поведении системы. Из-за этого могут возникнуть такие проблемы, как периодические стопоры сервера или спорадические медленные запросы. Их довольно трудно обнаружить.

Мы иногда запускаем множество эталонных тестов, чтобы исследовать или нагрузить определенные части сервера и таким образом лучше разобраться в их поведении. Хорошим примером является множество тестов, которые мы запускали на протяжении многих лет, чтобы понять поведение сброса данных InnoDB. Нашей целью была разработка алгоритмов очистки для различных рабочих нагрузок и типов оборудования. Часто бывает так, что мы тщательно тестируем различные настройки, чтобы понять, какой эффект они производят, и найти способы их оптимизации. Но это не так-то просто сделать — процесс может занять много дней или недель, и кроме того, это не подходит для большинства ситуаций, поскольку такое туннельное видение конкретной части сервера часто затмевает другие проблемы. Например, иногда мы обнаруживаем, что определенные комбинации параметров обеспечивают лучшую производительность в предельных случаях, однако эти параметры не подходят для промышленного использования из-за таких факторов, как потеря огромного объема памяти или оптимизация пропускной способности при полном игнорировании влияния на восстановление после сбоев.

Мы советуем подготовить комплект эталонных тестов перед тем, как приступать к настройке сервера. Вам нужно что-то, что отражает общую рабочую нагрузку и включает в себя крайние случаи, такие как очень большие и сложные запросы. Обычно хорошие результаты дает воспроизведение реальной рабочей нагрузки при обработке реальных данных. Обнаружив конкретную проблему, скажем медленно выполняющийся одиночный запрос, вы можете попробовать оптимизировать данный случай, но при этом рискуете непреднамеренно негативно воздействовать на другие запросы.

Самый лучший подход к делу — изменять не более одной-двух переменных за раз и после каждой такой модификации в течение длительного времени прогонять тесты. Иногда результаты бывают удивительными: вот вы немного изменили переменную,

и результаты улучшились, потом еще чуть-чуть изменили — и производительность резко упала. Если после такого изменения быстродействие падает, то, возможно, вы запросили слишком большой объем ресурса. Например, речь может идти о чрезмерном размере памяти для буфера, который часто выделяется и освобождается. Не исключено также, что возникло несоответствие между MySQL и операционной системой или оборудованием. Так, мы обнаружили, что оптимальное значение переменной `sort_buffer_size` может зависеть от способа работы кэша процессора, а при настройке переменной `read_buffer_size` нужно учитывать конфигурацию упреждающего чтения сервера и общие настройки подсистемы ввода/вывода. Больше — не всегда лучше, а может оказаться и намного хуже. Кроме того, некоторые переменные зависят от других, и в полной мере освоить все это можно только на опыте и при условии понимания архитектуры системы.

Когда эталонное тестирование полезно

Мы не рекомендовали применять эталонное тестирование. Однако из этого совета есть несколько исключений. Иногда все-таки стоит запускать несколько итеративных эталонных тестов, хотя обычно не в контексте тонкой настройки сервера. Приведем несколько примеров.

- Если вы планируете осуществить крупные инвестиции, такие как покупка нескольких новых серверов, можете запустить эталонные тесты, чтобы понять, каковы ваши потребности в оборудовании. Контекст здесь — планирование емкости, а не тонкая настройка сервера. В частности, нам нравится проводить эталонное тестирование с разным объемом памяти, выделенной буферному пулу InnoDB, что помогает нарисовать кривую памяти, показывающую, сколько памяти действительно необходимо и как это влияет на требования к системам хранения.
- Если вы хотите понять, сколько времени потребуется InnoDB, чтобы восстановиться после сбоя, можете неоднократно настраивать реплику, намеренно ее обрушивать и выполнять эталонное тестирование того, сколько времени требуется InnoDB на восстановление после перезапуска. Контекст здесь — планирование высокой работоспособности.
- Для приложений с большим объемом чтения может быть целесообразно зафиксировать все запросы с помощью журнала медленных запросов (или из трафика TCP с помощью утилиты pt-query-digest), далее применить утилиту pt-log-player для их повторения с включенным полным журналированием медленных запросов, а затем проанализировать полученный журнал с помощью pt-query-digest. Это позволяет увидеть, как различные типы запросов выполняются при различных настройках оборудования, программного обеспечения и сервера. Например, мы когда-то помогли клиенту оценить изменения производительности при переходе на сервер с гораздо бóльшим объемом памяти, но с медленными жесткими дисками. Большинство запросов стали выполняться быстрее, но некоторые аналитические запросы замедлялись, поскольку оставались связанными с вводом/выводом. В данном случае контекстом является сравнение рабочей нагрузки.

Чего делать не следует

Прежде чем начать настраивать сервер, мы хотели бы призвать вас избегать некоторых распространенных приемов, которые считаем рискованными или вредными. Внимание: впереди непечатные выражения!

Во-первых, вы не должны производить тонкую настройку по коэффициенту. Классический пример коэффициента тонкой настройки — это эмпирическое правило, гласящее, что коэффициент попаданий в кэш ключей должен быть выше, чем некое процентное соотношение, а вам следует увеличивать размер кэша, если коэффициент попадания слишком низкий. Это совершенно неправильный совет. Независимо от того, что кто-то вам сказал, *коэффициент попаданий в кэш не имеет никакого отношения к тому, слишком большой кэш или нет.* Начнем с того, что коэффициент попадания зависит от рабочей нагрузки: некоторые рабочие объемы просто некэшируемы независимо от того, насколько велик кэш, к тому же попадания в кэш бессмысленны по причинам, которые мы объясним позже. Иногда бывает, что, когда кэш слишком мал, коэффициент попадания низок, а увеличение размера кэша его увеличивает. Однако это случайная взаимосвязь, ничего не говорящая о производительности или правильном определении размера кэша.

Проблема с взаимосвязями, которые иногда кажутся истинными, заключается в том, что люди начинают верить, что они всегда будут таковыми. Администраторы баз данных Oracle отказались от настройки на основе коэффициентов много лет назад, и жаль, что администраторы баз данных MySQL не последовали их примеру[1]. Еще больше мы жалеем, что люди пишут, как они считают, скрипты тонкой настройки, кодифицирующие эти опасные приемы, и учат пользоваться ими тысячи людей. Таким образом, мы приходим ко второй рекомендации относительно того, чего делать не следует: не используйте скрипты тонкой настройки! Некоторые из них настолько популярны, что их можно найти в Интернете. Лучше всего их игнорировать[2].

Мы также предлагаем вам избегать словосочетания «тонкая настройка», которое часто использовали в нескольких последних разделах. Вместо этого мы предпочитаем использовать термины «конфигурация» или «оптимизация» (до тех пор пока вы на самом деле это делаете, см. главу 3). Когда мы видим словосочетание «тонкая настройка», воображение рисует новичка, который настраивает сервер и смотрит, что происходит. В предыдущем разделе мы рекомендовали оставить эту методику для

---

[1] Если вы не уверены, что тонкая настройка по коэффициенту плоха, прочитайте, пожалуйста, книгу Optimizing Oracle Performance Кэри Миллсапа (Cary Millsap) (издательство O'Reilly). В ней есть приложение, посвященное инструменту, который может искусственно генерировать любой коэффициент попадания в кэш по вашему желанию независимо от того, как плохо работает система. Естественно, это все сделано, чтобы показать, насколько бесполезным является коэффициент.

[2] Исключение: мы обслуживаем бесплатный (хороший) онлайновый инструмент конфигурации на http://tools.percona.com. Да, мы предвзяты.

тех, кто изучил внутреннее устройство сервера. Тонкая настройка вашего сервера может оказаться всего лишь тратой времени.

Поиск в Интернете рекомендаций по разработке конфигурации тоже не всегда отличная мысль. Вы можете найти множество плохих советов в блогах, на форумах и т. д.[1] Многие эксперты делятся в Интернете своими знаниями, но не всегда можно точно определить их квалификацию. Разумеется, мы не можем дать объективные рекомендации о том, где найти настоящих экспертов. Но можем сказать, что надежные, авторитетные поставщики услуг MySQL — это в целом более безопасная ставка, чем результаты простого поиска в Интернете, поскольку люди, у которых есть счастливые клиенты, вероятно, делают что-то правильно. Однако даже их советы может быть опасно применять, не разобравшись и не проводя тестирование, поскольку они могут относиться к ситуации, отличающейся от вашей, и неясным для вас деталям.

Наконец, не верьте в правильность популярной формулы потребления памяти — да, той самой, которую сама MySQL выдает при сбое. (Мы не будем ее здесь приводить.) Эта формула сохранилась с древнейших времен. Она не является надежным или хотя бы полезным способом понять, сколько памяти MySQL может использовать в худшем случае. Кроме того, вы можете найти вариации этой формулы в Интернете. Они также ошибочны, даже если в них добавлены факторы, отсутствующие в исходной формуле. По правде говоря, вы не можете установить верхнюю границу потребления памяти MySQL. Она не регулируется жестко сервером базы данных, который управляет распределением памяти. Это можно очень просто доказать, войдя на сервер и запустив несколько запросов, которые потребляют много памяти:

```
mysql> SET @crash_me_1 := REPEAT('a', @@max_allowed_packet);
mysql> SET @crash_me_2 := REPEAT('a', @@max_allowed_packet);
# ... run a lot of these ...
mysql> SET @crash_me_1000000 := REPEAT('a', @@max_allowed_packet);
```

Запустите все это в цикле, каждый раз создавая новые переменные, и в итоге у сервера закончится память и он рухнет! К тому же эти действия не требуют никаких привилегий.

Утверждения, которые мы попытались проиллюстрировать в этом разделе, не раз делали нас непопулярными среди людей, которые обвиняли нас в высокомерии, в том, что мы пытаемся дискредитировать других и выставляем себя единственными авторитетами. Некоторые утверждали, что так мы стремимся продвигать свои услуги. Мы не намерены быть эгоистами. Просто слышали так много плохих советов, которые кажутся дельными, если вы недостаточно опытны, и столько раз помогали разгребать завалы, что считаем своей задачей развенчать несколько мифов и порекомендовать нашим читателям не доверять всем экспертам подряд. В дальнейшем постараемся избегать непарламентских выражений.

---

[1] Вопрос: «Как формируется запрос?» Ответ: «Необхадима пазвать админа, который убьет их запросы, патаму што эти запросы не могут окрысится в атвет» *(сохранена авторская орфография)*.

Создание конфигурационного файла MySQL

Как мы предупреждали в самом начале главы, у нас нет одного самого лучшего конфигурационного файла, скажем, для сервера с четырьмя процессорами, 16 Гбайт оперативной памяти и 12 жесткими дисками, который годился бы на все случаи жизни. Вам все-таки придется разработать собственные конфигурации, поскольку даже при наличии неплохой отправной точки все сильно зависит от того, как используется конкретное оборудование.

Не все скомпилированные по умолчанию настройки MySQL хороши, но большинство из них вполне пригодны. Они рассчитаны на умеренное потребление ресурсов, поскольку предполагается, что MySQL не единственная программа, работающая на сервере. В конфигурации по умолчанию используется ровно столько ресурсов, сколько необходимо для запуска MySQL и выполнения простых запросов к небольшому объему данных. Если объем хранимой в базах информации превышает несколько мегабайтов, то, безусловно, параметры необходимо настраивать.

Можно начать с одного из образцов файлов конфигурации, включенных в дистрибутив сервера MySQL, однако учитывая имеющиеся у них проблемы. Например, в них часто встречаются закомментированные параметры, так что вам может прийти в голову мысль о необходимости выбора подходящих значений и их раскомментировании (это немного напоминает конфигурационный файл Apache). Кроме того, там встречается множество скучных комментариев, объясняющих варианты выбора, но эти объяснения не всегда корректны. Некоторые из этих вариантов вообще нельзя применить к популярным операционным системам! Наконец, приведенные образцы полностью устарели с точки зрения современного оборудования и рабочих нагрузок.

На протяжении многих лет эксперты по MySQL дискутировали о том, как устранить эти проблемы, но воз и ныне там. Наша рекомендация звучит следующим образом: не используйте в качестве отправной точки ни эти файлы, ни образцы, которые поставляются вместе с пакетами вашей операционной системы. Лучше начинать с чистого листа.

Именно этим мы и займемся в этой главе. На самом деле то, что MySQL так хорошо настраивается, — это ее недостаток: вам кажется, что следует потратить уйму времени на настройку, в то время как большинство параметров в порядке, просто заданы по умолчанию. И самое лучшее, что вы можете сделать, — оставить их в покое и забыть. Поэтому мы создали для этой книги работоспособный минимальный примерный конфигурационный файл, который можно использовать в качестве отправной точки для собственных серверов[1]. Вам придется выбрать значения только для нескольких

[1] Обратите внимание на то, что в новых версиях MySQL некоторые параметры удалены, помечены как нежелательные и изменены. Для получения более подробной информации посмотрите соответствующие руководства.

параметров — мы объясним это чуть позже в текущей главе. Наш базовый файл выглядит следующим образом:

```
[mysqld]
# GENERAL datadir                    = /var/lib/mysql
socket                               = /var/lib/mysql/mysql.sock
pid_file                             = /var/lib/mysql/mysql.pid
user                                 = mysql
port                                 = 3306
storage_engine                       = InnoDB
# INNODB
innodb_buffer_pool_size              = <value>
innodb_log_file_size                 = <value>
innodb_file_per_table                = 1
innodb_flush_method                  = O_DIRECT
# MyISAM
key_buffer_size                      = <value>
# LOGGING
log_error                            = /var/lib/mysql/mysql-error.log
log_slow_queries                     = /var/lib/mysql/mysql-slow.log
# OTHER
tmp_table_size                       = 32M
max_heap_table_size                  = 32M
query_cache_type                     = 0
query_cache_size                     = 0
max_connections                      = <value>
thread_cache_size                    = <value>
table_cache_size                     = <value>
open_files_limit                     = 65535
[client] socket                      = /var/lib/mysql/mysql.sock
port                                 = 3306
```

Этот файл может показаться *слишком* минималистичным по сравнению с тем, что вы привыкли видеть[1], но на самом деле он включает в себя даже больше, чем нужно многим. Есть несколько других типов параметров конфигурации, которые вы, вероятно, будете использовать, например двоичное журналирование. Мы рассмотрим их позже в этой и других главах.

Первым делом мы сконфигурировали местоположение данных. Для этого выбрали `/var/lib/mysql`, поскольку это популярное место для большинства вариантов Unix. Но нет ничего плохого и в выборе другого места — решать вам. Туда же мы поместили файл PID, однако многие операционные системы захотят поместить его в `/var/run`. Все в порядке, нам просто требовалось установить что-то для этих настроек. Кстати, не позволяйте сокету и PID-файлу располагаться в соответствии с установленными по умолчанию настройками сервера: в некоторых версиях MySQL есть ряд ошибок, которые из-за этого могут вызвать проблемы. Лучше установить их местоположение явно. (Мы не советуем выбирать разное местоположение, рекомендуем лишь

[1] Вопрос: «Где настройки размера буфера сортировки и размера буфера чтения?» Ответ: «Они занимаются своим делом. Оставьте для них значения по умолчанию, если не можете доказать, что эти значения недостаточно хороши».

убедиться, что в файле `my.cnf` их местоположение упомянуто явно, поэтому оно не будет меняться и создавать проблемы при обновлении сервера.)

Кроме того, мы определили, что `mysqld` должен работать как аккаунт пользователя `mysql` в операционной системе. Вам потребуется убедиться, что этот аккаунт существует и что ему принадлежит каталог с данными. Для порта установлено значение по умолчанию `3306`, но иногда его стоит поменять.

В качестве подсистемы хранения по умолчанию мы выбрали InnoDB, и это стоит пояснить. Мы полагаем, что InnoDB — наиболее удачный выбор в большинстве ситуаций, но не всегда. Например, какое-то стороннее программное обеспечение может считать, что по умолчанию используется MyISAM, и создаст таблицы без указания подсистемы. Это может привести к сбою программного обеспечения, если, например, предполагается, что в базе данных разрешено создавать полнотекстовые индексы. Кроме того, подсистема хранения по умолчанию используется для принудительно созданных временных таблиц, что может потребовать от сервера довольно много дополнительной работы. Если вы хотите, чтобы постоянные таблицы использовали InnoDB, а любые временные таблицы — MyISAM, то должны явно указать подсистему в команде `CREATE TABLE`.

В общем случае, если вы решили использовать подсистему хранения в качестве выбираемой по умолчанию, лучше всего настроить ее как значение по умолчанию. Многие пользователи считают, что они используют только определенную подсистему хранения, но затем обнаруживают другую подсистему, которая использовалась из-за того, что была задана по умолчанию.

Проиллюстрируем основы создания конфигурации с помощью InnoDB. В большинстве случаев все, что необходимо InnoDB для хорошей работы, — правильный размер буферного пула и размер файла журнала. Эти значения по умолчанию слишком малы. Остальные настройки для InnoDB являются необязательными, хотя мы включили `innodb_file_per_table` для обеспечения удобства управления и большей гибкости. Установку размера файла журнала InnoDB мы обсудим позже в этой главе, как и параметр `innodb _flush_method`, присущий Unix.

Существует широко известное эмпирическое правило, которое гласит: размер буферного пула должен составлять 75–80 % от объема памяти сервера. Это еще одно случайное соотношение, которое, похоже, иногда работает нормально, но далеко не всегда. Лучше всего установить буферный пул примерно следующим образом.

1. Возьмите объем памяти сервера.
2. Вычтите немного памяти для операционной системы и, возможно, для других программ, если MySQL — это не единственное, что работает на сервере.
3. Вычтите еще немного памяти для потребностей памяти MySQL, так как она, например, использует различные буферы для операций запроса.
4. Вычтите достаточно памяти для файлов журнала InnoDB так, чтобы операционная система имела достаточно памяти для их кэширования или по крайней мере последних записей в них. (Этот совет относится к стандартной MySQL, в Percona

Server вы можете настроить файлы журналов так, чтобы они открывались с помощью команды `O_DIRECT`, в обход кэшей операционной системы.) Также стоит оставить некоторую свободную память для кэширования по крайней мере хвоста двоичных журналов, особенно если у вас есть подчиненные серверы, которые запаздывают из-за того, что иногда могут читать старые двоичные файлы журнала на главном сервере, вызывая определенное давление на его память.

5. Вычтите достаточно памяти для любых других буферов и кэшей, которые вы настраиваете в MySQL, таких как кэш ключей MyISAM или кэш запросов.
6. Разделите полученный результат на 105 %. Тем самым вы учтете приблизительные накладные затраты InnoDB на управление буферным пулом.
7. Округлите результат до разумного значения. Округление в меньшую сторону не сильно изменит ситуацию, но излишнее выделение памяти, скорее всего, будет неверным решением.

Мы довольно легкомысленно рассуждали об объемах выделяемой памяти — все-таки «немного для операционной системы» — это сколько? Тут могут быть разные варианты, и мы частично обсудим это чуть позже в этой главе и в остальной части книги. Вам нужно разобраться в своей системе и оценить, сколько памяти ей нужно для работы. Вот почему конфигурационные файлы на все случаи жизни невозможны. Опыт, а иногда и несложные вычисления будут вашими помощниками.

Приведем пример. Предположим, у вас есть сервер с 192 Гбайт памяти и вы хотите выделить его для MySQL и использовать только InnoDB, без кэша запросов. При этом ожидаете, что количество подключений к серверу будет невелико. Если в ваших файлах журнала всего 4 Гбайт, вы можете рассуждать так: «Я думаю, что для ОС и других потребностей в памяти MySQL должно быть достаточно 2 Гбайт или 5 % от общей памяти в зависимости от того, что больше. Вычитаем 4 Гбайт для файлов журнала, а все остальное используем для InnoDB». У нас получилось около 177 Гбайт, но, по всей видимости, стоит немного уменьшить это значение. Возможно, вы выделите около 168 Гбайт на буферный пул. Если на практике сервер работает с достаточным количеством нераспределенной памяти, вероятно, вы установите буферный пул больше, если существует возможность перезапуска его для какой-то другой цели.

Результат был бы совсем другим, если бы у вас было несколько таблиц MyISAM и, естественно, нужно было бы кэшировать их индексы. В Windows результат также сильно отличался бы от приведенного ранее, поскольку в этой операционной системе в большинстве версий MySQL возникают проблемы с использованием больших объемов памяти (хотя эта ситуация и улучшена в MySQL 5.5) или если по какой-то причине вы решили не использовать `O_DIRECT`.

Как видите, не так уж важно устанавливать точные настройки с самого начала. Лучше задать безопасное значение, которое больше значения по умолчанию, но ненамного, запустить сервер на некоторое время и посмотреть, сколько памяти он фактически использует. Эти параметры трудно предвидеть, поскольку использование памяти MySQL не всегда предсказуемо: оно может зависеть от таких факторов, как сложность запросов и параллелизм. При простой рабочей нагрузке потребности

в памяти MySQL невелики — около 256 Кбайт на соединение. Но сложные запросы, использующие временные таблицы, сортировку, хранимые процедуры и т. п., могут использовать намного больше ОЗУ.

Вот почему мы выбрали довольно безопасную стартовую точку. Вы можете видеть, что даже консервативная настройка для буферного пула InnoDB на самом деле составляет 87,5 % от общего объема ОЗУ сервера, то есть более 75 %. Именно поэтому мы и говорили, что использование простых коэффициентов — это неправильный подход.

Наша позиция такова: при настройке буферов памяти лучше допустить ошибку, оказавшись чересчур осторожными, чем сделать их слишком большими. Если вы зададите буферный пул на 20 % меньше, чем могли, то, скорее всего, не сильно ухудшите производительность — возможно, на несколько процентов. Если же установите его на 20 % больше, чем требуется, то, скорее всего, столкнетесь с гораздо более серьезными проблемами: подкачкой, переполнением диска или даже полным исчерпанием памяти и аварийным остановом.

Рассмотренный пример конфигурации InnoDB демонстрирует предпочитаемый нами подход к настройке сервера: понять, что в нем происходит, как это связано с настройками, а затем решить, что делать.

Изменения с течением времени

Необходимость в точной настройке буферов памяти MySQL с течением времени становится все менее важной. Когда у мощного сервера было 4 Гбайт памяти, мы много работали, чтобы сбалансировать ресурсы так, чтобы он мог обслуживать 1000 подключений. Это, как правило, требовало резервирования 1 Гбайт или около того на потребности MySQL, что составляло четверть общей памяти сервера и сильно влияло на размер буферного пула.

В настоящее время у сопоставимого сервера 144 Гбайт памяти, при этом число обычных подключений в большинстве приложений не очень изменилось, равно как и размер буферов для каждого соединения. Так что мы могли бы щедро зарезервировать 4 Гбайт памяти для MySQL, а это капля в море и незначительно влияет на размер буферного пула.

Большинство других параметров в примере вполне понятны, и какими они будут — зависит только от нашего решения. Некоторые из них мы рассмотрим в дальнейшем в данной главе. Как видите, мы включили журналирование, отключили кэш запросов и т. д. В этой главе мы также обсудим некоторые настройки безопасности и готовности к работе, которые могут быть очень полезными для повышения надежности сервера, предотвращения появления плохих данных и других проблем. Здесь мы не показали эти настройки.

Один из параметров, который мы сейчас объясним, — `open_files_limit`. Мы установили его максимально большим для типичной системы Linux. Открытие дескрипто-

ров файлов в современных операционных системах не требует высоких затрат. Если этот параметр недостаточно велик, вы увидите ошибку 24 Too many open files (Слишком много открытых файлов).

Последний раздел в файле конфигурации предназначен для клиентских программ, таких как mysql и mysqladmin, и просто объясняет им, как подключиться к серверу. Вы должны установить для клиентских программ значения, совпадающие с выбранными для сервера.

Проверка переменных состояния сервера MySQL. Иногда для того, чтобы лучше настроить параметры сервера, учитывающие фактическую рабочую нагрузку, в качестве исходных данных можно использовать вывод команды `SHOW GLOBAL STATUS`. Для достижения наилучших результатов обращайте внимание как на абсолютные значения, так и на то, как значения меняются со временем, особенно в пиковые и непиковые моменты. При этом стоит использовать снимки состояния. Кроме того, для просмотра изменений в переменных состояния каждые 60 секунд можно применять следующую команду:

```
$ mysqladmin extended-status -ri60
```

По мере объяснения различных настроек конфигурации мы будем часто ссылаться на изменения переменных состояния во времени. Обычно мы ожидаем, что вы будете изучать вывод команды, такой как приведена ранее. Другими полезными инструментами, которые могут обеспечить компактное отображение изменений счетчика статусов, являются pt-mext и pt-mysql-summary пакета Percona Toolkit.

Теперь, закончив приготовления, проведем для вас экскурсию по некоторым частям внутреннего устройства сервера, перемежая ее советами по настройке. Таким образом, вы получите базу, которая понадобится для выбора подходящих значений параметров конфигурации, когда мы позднее вернемся к образцу конфигурационного файла.

Настройка использования памяти

Правильное конфигурирование работы MySQL с памятью жизненно важно для достижения хорошей производительности. Распределение памяти вам придется настраивать почти наверняка. Потребляемая память в MySQL делится на две группы: подконтрольную и неподконтрольную вам. Вы не можете управлять тем, сколько памяти MySQL использует для запуска сервера, разбора запросов и внутренних целей, но тем, сколько ее расходуется на различные конкретные цели, управлять вполне можно. Правильно распорядиться подконтрольной вам памятью не так уж сложно, если понимать, что именно конфигурируется.

Как было показано ранее, настраивать использование памяти можно поэтапно.

1. Определить абсолютный верхний предел объема памяти, которую MySQL может использовать.
2. Определить, сколько памяти MySQL будет использовать на каждое соединение, например для буферов сортировки и временных таблиц.

3. Определить, сколько памяти нужно операционной системе для нормальной работы. Включить сюда память, необходимую для других программ, работающих на той же машине, например периодически выполняемых заданий.
4. Если это имеет смысл, отдать всю оставшуюся память под кэши MySQL, например под буферный пул InnoDB.

Мы рассмотрим эти шаги в последующих разделах, а затем более подробно обсудим требования к различным кэшам MySQL.

Сколько памяти может использовать MySQL

В любой конкретной системе существует верхний предел объема памяти, в принципе доступной MySQL. За точку отсчета следует принять объем физически установленной памяти. Если у сервера нет памяти, то и MySQL ее не сможет использовать.

Примите во внимание также ограничения, характерные для операционной системы и аппаратной архитектуры, например лимиты размера адресного пространства одного процесса в 32-разрядной ОС. Поскольку MySQL работает как один процесс с несколькими потоками, объем доступной ей памяти может быть серьезно ограничен. Скажем, в 32-разрядных ядрах Linux любой процесс может адресовать 2,5–2,7 Гбайт памяти. Выход за пределы адресного пространства очень опасен и может привести к аварийному останову MySQL. Сейчас это случается крайне редко, но было обыденностью в прежние годы.

Существует много зависящих от операционной системы параметров и странностей, которые следует учитывать. К ним относятся не только лимиты на ресурсы, выделяемые одному процессу, но и размер стека и другие настройки. Ограничения на размер одного выделяемого блока памяти может налагать и системная библиотека glibc. Например, невозможно присвоить параметру `innodb_buffer_pool` значение, превышающее 2 Гбайт, если это максимальный размер блока, который может быть выделен glibc за один подход.

Некоторые ограничения существуют даже для 64-разрядных серверов. Например, на 64-разрядном сервере размеры многих буферов, в частности буфера ключей, ограничены 4 Гбайт. Часть ограничений снята в MySQL 5.1. Максимальные значения каждой конфигурационной переменной документированы в руководстве по MySQL.

Сколько памяти нужно соединению

MySQL нуждается в небольшом объеме памяти просто для того, чтобы поддерживать соединение (поток) открытым. Кроме того, определенное количество памяти необходимо, чтобы выполнить запрос. Вы должны выделить достаточно ресурсов для выполнения запросов даже в периоды пиковой нагрузки. В противном случае запросы будут сидеть на голодном пайке, вследствие чего станут выполняться очень медленно или вообще будут завершаться с ошибкой.

Вообще полезно знать, сколько памяти MySQL потребляет в пиковые периоды, но в некоторых ситуациях потребление памяти неожиданно и резко возрастает, что делает любые прогнозы ненадежными. Один из примеров — подготовленные операторы, поскольку их может быть открыто сразу много. Другой пример — словарь данных InnoDB (подробнее об этом рассказывается в дальнейшем).

Пытаясь спрогнозировать пиковое потребление памяти, не обязательно предполагать наихудший сценарий. Например, если MySQL сконфигурирована из расчета не более 100 одновременных соединений, то теоретически возможно, что на всех 100 соединениях одновременно будут выполняться очень тяжелые запросы, но практически этого, скорее всего, не случится. Если вы установите параметр `myisam_sort_buffer_size` равным `256M`, то при худшем варианте развития событий потребуется по меньшей мере 25 Гбайт памяти, но такой уровень потребления крайне маловероятен. Запросы, которые используют множество больших временных таблиц или сложные хранимые процедуры, являются наиболее вероятными причинами высокого потребления памяти для каждого подключения.

Чем вести подсчеты для худшего случая, лучше понаблюдать за сервером в условиях реальной нагрузки и посмотреть, сколько же памяти он потребляет. Для этого взгляните на размер виртуальной памяти процесса. Во многих UNIX-системах этот показатель отображается в столбце `VIRT` таблицы, выдаваемой командой `top`, или в столбце `VSZ`, если вы пользуетесь командой `ps`. В следующей главе мы подробнее расскажем о том, как вести мониторинг потребления памяти.

Резервирование памяти для операционной системы

Для работы операционной системы, как и для выполнения запросов, необходимо отвести достаточно памяти. Лучшим свидетельством того, что в распоряжении ОС достаточно памяти, является отсутствие выгрузки страниц в файл подкачки на диске (см. следующую главу, где эта тема обсуждается более подробно).

Зарезервируйте для нужд операционной системы по меньшей мере 1–2 Гбайт или даже больше для машин с большим объемом памяти. Добавьте небольшой запас для страховки, а если периодически запускаются задачи, потребляющие необычно много памяти (к примеру, резервное копирование), можете сделать этот запас побольше. Не добавляйте память для кэшей операционной системы, поскольку они могут быть очень велики. Обычно ОС использует в этих целях всю свободную память (далее мы рассмотрим кэши отдельно от нужд операционной системы).

Выделение памяти для кэшей

Если сервер выделен исключительно для MySQL, то вся память, не отведенная для нужд операционной системы или обработки запросов, может быть использована в целях кэширования.

Для кэшей MySQL нужно больше памяти, чем для чего-либо другого. Кэши используются, чтобы избежать доступа к диску, который на несколько порядков медленнее, чем обращение к памяти. Операционная система может кэшировать некоторые данные в интересах MySQL (особенно в случае MyISAM), но и самой MySQL требуется много памяти.

Далее перечислены наиболее важные кэши, которые следует учитывать в большинстве случаев:

- буферный пул InnoDB;
- кэши операционной системы для файлов журналов InnoDB и данных MyISAM;
- кэши ключей MyISAM;
- кэш запросов;
- кэши, которые вы не можете настроить, например кэши операционной системы двоичных журналов и файлы определения таблиц.

Существуют и другие кэши, но они, как правило, потребляют не так много памяти. Кэш запросов мы подробно рассматривали в предыдущей главе, поэтому далее займемся кэшами, которые необходимы подсистемам хранения MyISAM и InnoDB для нормальной работы.

Сервер гораздо проще настроить, если используется только одна подсистема хранения. Если вы работаете исключительно с таблицами MyISAM, то InnoDB можно вообще отключить, а если только с таблицами InnoDB, то отвести под MyISAM минимум ресурсов (MySQL применяет таблицы типа MyISAM для внутренних нужд). Однако в ситуации, когда используется сразу несколько подсистем хранения, бывает очень трудно подобрать их правильное соотношение. Лучший подход, который мы можем предложить, — высказать некую обоснованную гипотезу, а затем посмотреть на работу сервера.

Буферный пул InnoDB

Если вы работаете преимущественно с таблицами InnoDB, то для буферного пула InnoDB, вероятно, потребуется больше памяти, чем для чего-либо другого. В буферном пуле InnoDB кэшируются не только индексы, там хранятся также сами данные, адаптивный хеш-индекс, буфер вставок, блокировки и другие внутренние структуры. В InnoDB буферный пул используется также для реализации отложенных операций записи и позволяет объединить несколько таких процедур, чтобы затем выполнить их последовательно. Короче говоря, работа InnoDB *очень сильно* зависит от буферного пула, поэтому памяти в нем должно быть достаточно. Обеспечить это можно, следуя приведенным ранее рекомендациям. Для мониторинга производительности и использования памяти в буферном пуле можно проанализировать переменные, выводимые командами `SHOW` и инструментами типа innotop.

Если у вас не так уж много данных и вы знаете, что их объем не будет быстро расти, то не нужно выделять слишком много памяти в буферный пул. Не стоит делать его

намного больше, чем размер таблиц и индексов, которые будут в нем храниться. Конечно, нет ничего плохого в перспективном планировании для быстро растущей базы данных, но иногда мы видим огромные буферные пулы с крошечным объемом данных. Это совсем не обязательно.

Большие буферные пулы вызывают некоторые проблемы, такие как длительное время выключение и «прогрева». Если в буферном пуле есть много «грязных» (модифицированных) страниц, InnoDB может потребоваться много времени для останова, поскольку она записывает «грязные» страницы в файлы данных при выключении. Можно принудительно выполнить процедуру быстрого останова, но тогда InnoDB потребуется больше времени на восстановление при повторном запуске, так что уменьшить суммарное время останова и запуска не получится. Если вы заранее знаете, что придется останавливать сервер, уменьшите значение переменной `innodb_max_dirty_pages_pct` во время выполнения, дождитесь, когда поток сброса очистит буферный пул, и начинайте останов. Следить за количеством «грязных» страниц можно с помощью серверной переменной состояния `Innodb_buffer_pool_pages_dirty` или утилиты innotop, которая периодически выполняет команду `SHOW INNODB STATUS`.

Уменьшение значения переменной `innodb_max_dirty_pages_pct` еще не гарантирует, что InnoDB будет хранить в буферном пуле меньше «грязных» страниц. Она лишь управляет порогом, при котором InnoDB перестает лениться. По умолчанию InnoDB сбрасывает «грязные» страницы на диск в отдельном фоновом потоке, который ради эффективности группирует операции записи и выполняет их последовательно. Такое поведение называется *ленивым*, поскольку InnoDB откладывает сброс «грязных» страниц из пула до того момента, когда понадобится место для других данных. Но если процент «грязных» страниц превысит заданный порог, то InnoDB начинает сбрасывать страницы на диск с максимально возможной скоростью, стремясь уменьшить их количество. InnoDB также перейдет в режим яростного сброса, когда в журналах транзакций останется недостаточно места, что является одной из причин возможного повышения производительности больших журналов.

Когда у вас есть большой буферный пул, особенно в сочетании с медленными дисками, серверу может потребоваться много времени (много часов или даже дней), чтобы «прогреться» после перезагрузки. В таких случаях вы можете воспользоваться предоставляемой Percona Server возможностью перезагрузки страниц после перезапуска. Это может сократить время «прогрева» до нескольких минут. В MySQL 5.6 эта возможность также будет реализована. Это особенно выгодно для реплик, которые несут дополнительные затраты на «прогрев» из-за однопоточного характера репликации.

Если у вас нет возможности использовать способ быстрого «прогрева» от Percona Server, можно сразу же после перезагрузки выполнить полнотабличное сканирование или сканирование индексов, чтобы загрузить индексы в буферный пул. Это грубое решение, но, как правило, все же лучше, чем ничего. Для выполнения данной задачи можно использовать параметр `init_file`. Вы можете поместить SQL-код

в файл, который запускается при запуске MySQL. Имя файла должно быть указано в параметре `init_file`, а файл может содержать несколько команд SQL, каждая из которых находится в отдельной строке (комментарии не допускаются).

Кэш ключей MyISAM

Кэш ключей MySQL часто называют также *буфером ключей*. По умолчанию существует только один такой буфер, но можно создать дополнительные. В отличие от InnoDB и некоторых других подсистем хранения MyISAM самостоятельно кэширует только индексы, но не данные (она позволяет операционной системе кэшировать данные). Если вы используете преимущественно MyISAM, то следует выделить много памяти под кэши ключей.

Наиболее важен параметр `key_buffer_size`. Память, не выделенная в нем, будет отведена под кэши операционной системы, в которых обычно хранятся данные, считанные из MYD-файлов MyISAM. В версии MySQL 5.0 для этого параметра установлено ограничение 4 Гбайт вне зависимости от архитектуры. В MySQL 5.1 допустимы бóльшие значения. Уточните этот показатель в документации к версии вашего сервера.

Когда вы решаете, сколько памяти выделить кэшу ключей, может быть полезно знать, сколько места на диске в настоящий момент используют индексы MyISAM. Не стоит делать буферы ключей больше, чем данные, которые они будут кэшировать. Вы можете запросить таблицы `INFORMATION_SCHEMA` и просуммировать столбец `INDEX_LENGTH`, чтобы узнать размер файлов, хранящих индексы:

```
SELECT SUM(INDEX_LENGTH) FROM INFORMATION_SCHEMA.TABLES WHERE ENGINE='MYISAM';
```

В системах на базе UNIX можно использовать следующую команду:

```
$ du -sch `find /path/to/mysql/data/directory/ -name "*.MYI"`
```

Насколько значительными должны быть заданы кэши ключей? Не больше, чем общий размер индекса, или от 25 до 50 % объема памяти, зарезервированного для кэшей операционной системы, в зависимости от того, какое значение меньше.

По умолчанию MyISAM кэширует все индексы в буфере ключей, подразумеваемом по умолчанию, но в ней разрешается создавать дополнительные именованные буферы. Это дает возможность хранить в памяти более 4 Гбайт индексных ключей. Чтобы создать буферы ключей `key_buffer_1` и `key_buffer_2` по 1 Гбайт каждый, добавьте в конфигурационный файл такие строчки:

```
key_buffer_1.key_buffer_size = 1G
key_buffer_2.key_buffer_size = 1G
```

Теперь имеется три буфера ключей: два созданы явно, а один — по умолчанию. Команда `CACHE INDEX` позволяет установить соответствие между таблицами и кэшами. Вот как можно указать, что MySQL должна использовать кэш `key_buffer_1` для индексов таблиц `t1` и `t2`:

```
mysql> CACHE INDEX t1, t2 IN key_buffer_1;
```

Теперь все блоки, прочитанные из индексов этих таблиц, MySQL будет помещать в указанный буфер. Можно также заранее загрузить индексы в кэш командой `LOAD INDEX`:

```
mysql> LOAD INDEX INTO CACHE t1, t2;
```

Все индексы, с которыми явно не сопоставлен буфер ключей, ассоциируются с буфером, заданным по умолчанию, при первом доступе к соответствующему `.MYI`-файлу.

Следить за производительностью и использованием буферов ключей можно с помощью информации, которую выводят команды `SHOW STATUS` и `SHOW VARIABLES`. Процент заполнения буфера вычисляется по следующей формуле:

```
100 - ((Key_blocks_unused * key_cache_block_size) * 100/key_buffer_size)
```

Если сервер не использует весь свой буфер ключей, проработав в течение длительного времени, можно подумать об уменьшении буфера.

Что можно сказать о коэффициенте попадания в кэш? Как мы объяснили ранее, его значение бесполезно. Например, разница между 99 и 99,9 % кажется совсем небольшой, но в действительности это десятикратное увеличение. Коэффициент попаданий зависит также от приложения: некоторые из них прекрасно работают при 95 %, тогда как другие могут постоянно обращаться к диску даже при 99,9 %. Если размер кэша выбран верно, то можно даже достичь уровня 99,99 %.

Эмпирически обычно полезнее другой показатель — количество *непопаданий* в кэш в секунду. Предположим, что имеется один жесткий диск, способный выполнять 100 операций произвольного чтения в секунду. Пять непопаданий за этот промежуток времени еще не сделают систему ввода/вывода узким местом, но при 80 возникнут проблемы. Данное значение вычисляется по формуле:

```
Key_reads/Uptime
```

Чтобы составить представление о текущей производительности, вычисляйте количество непопаданий несколько раз с интервалом 10–100 секунд. Следующая команда снимает показания каждые 10 секунд:

```
$ mysqladmin extended-status -r -i 10 | grep Key_reads
```

Помните, что для файлов данных, которые зачастую больше индексов, MyISAM пользуется кэшем операционной системы. Поэтому разумно оставлять для этого кэша больше памяти, чем для кэшей ключей. Даже если у вас достаточно памяти для кэширования всех индексов, а коэффициент непопадания в кэш очень низок, непопадания в кэш, когда MyISAM пытается читать из файлов данных (а не из файлов индекса!), происходят на уровне операционной системы, невидимом для MySQL. Таким образом, у вас может быть много непопаданий в кэш файлов данных независимо от коэффициента непопадания в кэш индексов.

И наконец, даже если у вас вообще нет таблиц типа MyISAM, все равно следует выделить с помощью параметра `key_buffer_size` хотя бы небольшой объем памяти, скажем, 32 Мбайт. Иногда MySQL использует MyISAM-таблицы для внутренних

целей. Например, так устроены временные таблицы, создаваемые при обработке запросов с фразой `GROUP BY`.

Размер блока ключей MyISAM. Размер блока ключей важен (особенно когда в рабочей нагрузке превалируют операции записи) из-за способа взаимодействия между MyISAM, кэшем ОС и файловой системой. Если размер блока ключей слишком мал, то можно столкнуться с феноменом *записи после чтения*, то есть с такими операциями записи, которые операционная система не может выполнить, не прочитав предварительно какие-то данные с диска. Покажем, как может возникать этот феномен, предполагая, что размер страницы операционной системы равен 4 Кбайт (типично для архитектуры x86), а размер блока ключей равен 1 Кбайт.

1. MyISAM запрашивает блок ключей размером 1 Кбайт с диска.
2. Операционная система считывает страницу данных размером 4 Кбайт с диска, кэширует ее, а затем передает MyISAM затребованный 1 Кбайт.
3. Операционная система отбрасывает закэшированные данные, замещая их какими-то другими.
4. MyISAM модифицирует блок ключей размером 1 Кбайт и просит операционную систему записать его обратно на диск.
5. Операционная система считывает ту же самую страницу размером 4 Кбайт с диска в свой кэш, модифицирует в ней тот самый 1 Кбайт, который изменил MyISAM, и записывает все 4 Кбайт обратно на диск.

Феномен записи после чтения возникает на шаге 5, когда MyISAM просит операционную систему записать только часть страницы размером 4 Кбайт. Если бы размер блока MyISAM был равен размеру страницы ОС, то чтения на шаге 5 можно было бы избежать[1].

К сожалению, в MySQL 5.0 и более ранних версиях невозможно конфигурировать размер блока ключей. Однако, начиная с MySQL 5.1, можно избежать записи после чтения, сделав размер блока ключей MyISAM равным размеру страницы операционной системы. Величиной блока ключей управляет переменная `myisam_block_size`. Можно также задать размер блока для каждого ключа в отдельности с помощью параметра `KEY_BLOCK_SIZE` в командах `CREATE TABLE` и `CREATE INDEX`. Поскольку все ключи хранятся в одном и том же файле, то на самом деле нужно, чтобы для всех ключей размер блока был не меньше размера страницы операционной системы. Это позволяет избежать проблем с выравниванием, которые все равно могут привести к феномену записи после чтения (например, если для одного ключа используются блоки размером 1 Кбайт, а для другого — 4 Кбайт, то границы четырехкилобайтного блока могут не совпадать с границами страницы операционной системы).

---

[1] Теоретически, если бы можно было гарантировать, что исходные 4 Кбайт данных все еще находятся в кэше операционной системы, чтение было бы не нужно. Но вы не можете контролировать, какие блоки ОС оставляет в кэше. Получить сведения о содержимом кэша позволяет инструмент fincore, который можно скачать со страницы http://net.doit.wisc.edu/~plonka/fincore/.

Кэш потоков

Кэш потоков содержит потоки, которые в данный момент не ассоциированы ни с одним соединением, но готовы обслуживать новые соединения. Если в кэше есть поток и поступает запрос на создание нового соединения, то MySQL забирает поток из кэша и передает его создаваемому соединению. Когда соединение закрывается, MySQL возвращает поток в кэш, если там есть место. Если места нет, поток уничтожается. Пока в кэше есть свободные потоки, MySQL отвечает на запросы об открытии соединения очень быстро, поскольку ей не нужно создавать новый поток для обслуживания соединения.

Переменная `thread_cache_size` определяет максимальное количество потоков в этом кэше. Настраивать ее нужно лишь в случае, когда сервер получает очень много запросов на открытие соединений. Чтобы проверить, достаточен ли размер кэша, понаблюдайте за переменной состояния `Threads_created` `status`. Мы обычно стараемся делать кэш потоков настолько большим, чтобы в каждую секунду создавалось не более десяти новых потоков, но часто без труда удается снизить этот показатель, так что в секунду будет создаваться менее одного потока.

Правильный подход состоит в том, чтобы, понаблюдав за переменной `Threads_connected`, попытаться установить значение `thread_cache_size` достаточно большим для поглощения типичных флуктуаций рабочей нагрузки. Например, если `Threads_connected` обычно изменяется от 100 до 120, то можно сделать размер кэша равным 20. Если она изменяется от 500 до 700, то кэша размером 200 будет достаточно. Мы рассуждаем следующим образом: при 700 соединениях в кэше, вероятно, нет потоков, при 500 соединениях в кэше имеется 200 потоков, готовых к использованию, как только нагрузка вновь возрастет до 700.

В большинстве случаев нет необходимости делать кэш потоков очень большим, однако и уменьшение его сверх меры не дает заметной экономии памяти. Поток, находясь в кэше или будучи в состоянии сна, обычно занимает около 256 Кбайт. Это совсем немного по сравнению с памятью, потребляемой потоком во время обработки запроса. В общем случае старайтесь поддерживать размер кэша на уровне, при котором переменная `Threads_created` не увеличивается слишком часто. Но если при таком условии объем кэша становится слишком велик (порядка нескольких тысяч), то лучше сделать его поменьше, так как некоторые операционные системы плохо справляются с чрезмерно большим количеством потоков, даже если большинство из них спит.

Кэш таблиц

Концептуально кэш таблиц похож на кэш потоков, но хранятся в нем объекты, представляющие таблицы. Каждый такой объект содержит разобранный `.frm`-файл, описывающий таблицу, и некоторые другие данные. Что именно содержится в объекте, зависит от подсистемы хранения. Например, в MyISAM там находятся данные таблиц и/или дескрипторы файла индекса. Для объединенной таблицы в кэше, как

правило, хранятся несколько файловых дескрипторов, поскольку она может быть составлена из многих таблиц.

Кэш таблиц облегчает повторное использование ресурсов. Например, когда запрос обращается к таблице типа MyISAM, MySQL может вернуть файловый дескриптор из объекта в кэше. Это действительно позволяет избежать затрат на открытие файлового дескриптора, однако эти затраты не столь велики, как вы думаете. Открытие и закрытие дескрипторов файлов происходит очень быстро в локальном хранилище — сервер легко может проделать это миллион раз в секунду (хотя для сетевого хранилища это не так). Кроме того, кэш таблиц позволяет избежать ряда операций ввода/вывода, необходимых для того, чтобы пометить MyISAM-таблицу признаком in use (используется) в заголовках индекса[1].

Архитектура кэша таблиц — это одна из тех областей, в которых разделение между сервером и подсистемами хранения неочевидно, так уж сложилось исторически. Кэш таблиц не так уж важен для InnoDB, которая полагается на него в меньшей степени (скажем, файловых дескрипторов там нет, для этой цели у InnoDB есть собственный кэш таблиц). Однако даже InnoDB выигрывает от кэширования разобранных `.frm`-файлов.

В версии MySQL 5.1 кэш таблиц разделен на две части: кэш открытых таблиц и кэш определений таблиц (для их конфигурирования служат переменные `table_open_cache` и `table_definition_cache`). Таким образом, определения таблиц (разобранные `.frm`-файлы) отделены от других ресурсов, таких как файловые дескрипторы. Открытые таблицы по-прежнему локальны для соединения, но определения таблиц глобальны и могут совместно использоваться всеми соединениями. В общем случае переменной `table_definition_cache` можно присвоить значение, достаточно большое для кэширования определений всех таблиц. Если количество таблиц не исчисляется десятками тысяч, то это, наверное, проще всего.

Если переменная состояния `Opened_tables` велика или постоянно растет, то переменную `table_cache` (или `table_open_cache` в версии MySQL 5.1) следует увеличить. Однако обратите внимание на то, что этот счетчик увеличивается, когда вы создаете и удаляете временные таблицы, поэтому, если вы часто это делаете, счетчик никогда не перестанет увеличиваться.

Единственный недостаток избыточно большого кэша таблиц заключается в том, что при наличии очень большого количества MyISAM-таблиц может замедлиться останов сервера, поскольку необходимо сбросить на диск блоки ключей и пометить все

[1] Концепция открытой таблицы может сбить с толку. MySQL считает, что таблица открывается каждый раз, когда к ней производятся обращения из одновременно выполняемых запросов и даже из одного запроса, если таблица встречается в нем более одного раза, например в случае подзапроса или рефлексивного соединения. В индексных файлах MyISAM имеется счетчик, который MyISAM увеличивает при каждом открытии и уменьшает при закрытии. Это позволяет подсистеме хранения определить, что таблица не была закрыта корректно: это так, если при первом открытии таблицы счетчик отличен от нуля.

таблицы как неоткрытые. По той же самой причине команда `FLUSH TABLES WITH READ LOCK` может выполняться довольно долго.

Строго говоря, алгоритмы, проверяющие кэш таблиц, не очень эффективны (подробнее об этом позже).

Если вы получаете сообщение об ошибке, в котором говорится, что MySQL не может открыть новые файлы (понять, что означает код ошибки, поможет утилита perror), то, возможно, следует увеличить количество файлов, которые MySQL может держать открытыми. Для этого предназначена серверная переменная `open_files_limit` в файле `my.cnf`.

Кэши потоков и таблиц потребляют не так уж много памяти, а польза от них велика, поскольку они экономят ресурсы. Хотя создание потока и открытие новой таблицы — не очень затратные операции по сравнению с тем, что еще приходится делать MySQL, накладные затраты могут суммироваться. Таким образом, кэширование потоков и таблиц способно повысить эффективность.

Словарь данных InnoDB

В подсистеме хранения InnoDB имеется отдельный кэш, который называется *кэшем определений таблиц* или *словарем данных*, в текущих версиях MySQL он не допускает конфигурирования. Открывая таблицу, InnoDB помещает соответствующий ей объект в словарь данных. На одну таблицу может отводиться 4 Кбайт или больше (хотя в версии MySQL 5.1 требуется гораздо меньше памяти). Таблицы не удаляются из словаря данных при закрытии.

В результате со временем сервер может столкнуться с утечкой памяти из-за постоянно увеличивающегося количества записей в кэше словаря. Это не совсем утечка памяти — в данном случае не реализовано никакого истечения кэша. Обычно это становится проблемой, только когда у вас множество (тысячи или десятки тысяч) больших таблиц. Если это стало проблемой, можете использовать Percona Server, который имеет возможность ограничить размер словаря данных, удалив неиспользуемые таблицы. Аналогичная функция есть в еще не выпущенной MySQL 5.6.

Еще одной проблемой, связанной с производительностью, является вычисление статистики для таблиц во время их первого открытия, поскольку требует выполнения довольно большого числа операций ввода/вывода. В отличие от MyISAM InnoDB не хранит статистику в таблицах постоянно, а высчитывает заново при каждом запуске и после этого, по истечении различных интервалов или когда происходят особые события (изменения в содержимом таблицы, запросы к `INFORMATION_SCHEMA` и т. д.). Если в вашей базе много таблиц, то на полную загрузку и «прогрев» сервера может уйти несколько часов, в течение которых он только и будет делать, что ждать завершения последовательных операций ввода/вывода. Можете включить параметр `innodb_use_sys_stats_table` в Percona Server (аналогичный, под названием `innodb_analyze_is_persistent`, есть в MySQL 5.6), чтобы постоянно сохранять статистику на диске и решить эту проблему.

Даже после запуска операции статистики InnoDB могут влиять на сервер и отдельные запросы. Чтобы избежать трудоемких обновлений статистики таблиц, можно отключить параметр `innodb_stats_on_metadata`. Это особенно важно, когда такие инструменты, как IDE, запрашивают таблицы `INFORMATION_SCHEMA`.

При использовании параметра `innodb_file_per_table` (будет описан в дальнейшем) действует отдельное ограничение на количество одновременно открытых `.ibd`-файлов. Оно контролируется подсистемой хранения InnoDB, а не сервером и задается параметром `innodb_open_files`. InnoDB открывает файл не так, как MyISAM. Если MyISAM хранит файловые дескрипторы открытых таблиц в кэше, то в InnoDB нет прямой связи между открытыми таблицами и открытыми файлами. Эта подсистема использует один глобальный файловый дескриптор для каждого `.ibd`-файла. Если вы можете себе это позволить, то лучше присвоить параметру `innodb_open_files` настолько большое значение, чтобы сервер мог держать все `.ibd`-файлы открытыми одновременно.

Настройка ввода/вывода в MySQL

Ряд конфигурационных параметров управляет тем, как MySQL синхронизирует данные в памяти и на диске и выполняет восстановление. Эти параметры могут существенно повлиять на производительность, так как относятся к весьма затратным операциям ввода/вывода. Кроме того, они определяют компромисс между производительностью и защитой данных. В общем случае стремление записывать данные на диск немедленно и постоянно очень затратно. Если вы готовы пойти на риск, обусловленный тем, что запись на диск еще не обеспечивает постоянного хранения, то можно будет повысить степень конкурентности и/или сократить время ожидания ввода/вывода. Но определять допустимый уровень риска вам придется самостоятельно.

InnoDB

Подсистема хранения InnoDB позволяет управлять не только способом восстановления, но и тем, как открываются файлы и сбрасываются на диск данные кэшей, что сильно влияет на восстановление и общую производительность. Процесс восстановления в InnoDB полностью автоматический и обязательно выполняется в момент запуска InnoDB, хотя у вас остается возможность повлиять на то, какие действия при этом предпринимаются. Даже если оставить в стороне вопрос о восстановлении и считать, что никаких аварий не было и вообще все нормально, у вас в любом случае остается множество возможностей для конфигурирования InnoDB. В этой подсистеме имеется сложная цепочка буферов и файлов, спроектированная так, чтобы добиться высокой производительности и гарантировать свойства ACID. При этом каждое звено этой цепочки конфигурируемо. На рис. 8.1 показаны все файлы и буферы.

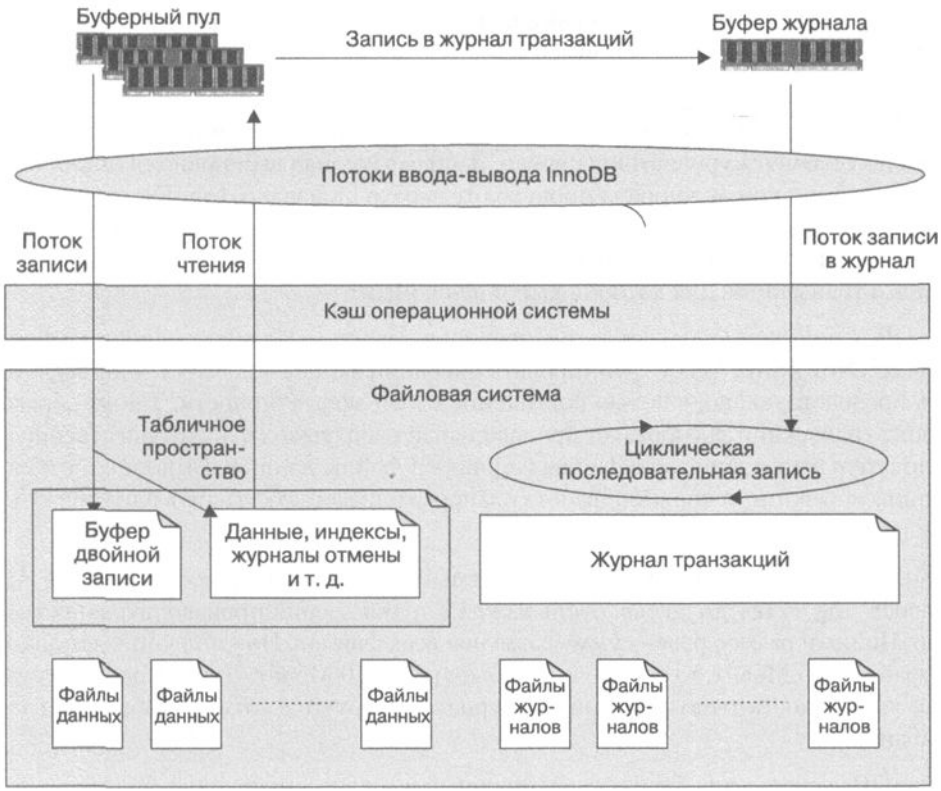


Рис. 8.1. Буферы и файлы подсистемы хранения InnoDB

Из наиболее важных настроек, которые имеет смысл изменять, отметим размер журнала InnoDB, способ сброса журнального буфера на диск и то, как выполняется ввод/вывод.

Журнал транзакций InnoDB

В InnoDB журнал служит для того, чтобы уменьшить затраты на коммит транзакций. Вместо сброса пула буферов на диск после коммита каждой транзакции InnoDB записывает транзакции в журнал. Изменения данных и индексов, произведенные внутри транзакции, часто относятся к различным местам в табличном пространстве, поэтому для их сброса на диск требуются произвольные операции ввода/вывода. InnoDB предполагает использование обычных дисков, где произвольный ввод/вывод намного затратнее, чем последовательный, из-за времени, которое требуется для поиска нужного местоположения на диске и ожидания его поворота до нужного участка.

В InnoDB журнал используется для превращения произвольного ввода/вывода в последовательный. После того как запись в журнал произведена, транзакцию можно

считать долговечной, даже если изменения еще не записаны в файлы данных. Если случится какая-то авария (например, пропадет питание), то InnoDB сможет воспроизвести журнал и восстановить закоммиченные транзакции.

Разумеется, InnoDB в конечном итоге должна записать изменения в файлы данных, поскольку размер журнала фиксирован. Запись в журнал выполняется циклически: по достижении конца журнала происходит переход в начало. InnoDB не может затереть запись журнала, если соответствующие ей изменения еще не были внесены в файлы данных, поскольку при этом была бы уничтожена единственная долговечная запись о транзакции, для которой выполнен коммит.

InnoDB использует фоновый поток для упорядоченного сброса изменений в файлы данных. Этот поток умеет группировать операции записи так, чтобы они выполнялись последовательно, с целью повышения своей эффективности. Таким образом, журнал транзакций преобразует произвольный ввод/вывод в преимущественно последовательный с записью в файлы журнала и файлы данных. Выполнение сброса в фоновом режиме ускоряет обработку запроса и помогает сгладить влияние всплесков нагрузки на подсистему ввода/вывода.

Полный размер файлов журнала, задаваемый параметрами `innodb_log_file_size` и `innodb_log_files_in_group`, очень важен с точки зрения производительности записи. Полный размер равен сумме размеров всех файлов. По умолчанию создаются два файла по 5 Мбайт, то есть полный объем равен 10 Мбайт. Для высокой нагрузки этого явно недостаточно. Для файлов журналов требуются сотни мегабайт или даже гигабайт.

В InnoDB несколько файлов образуют единый циклический журнал. Обычно не требуется изменять принимаемое по умолчанию количество журналов, настраивается лишь размер каждого файла. Чтобы изменить размер файла журнала, штатно остановите MySQL, переместите старые журналы в другое место, измените конфигурацию и перезапустите сервер. Очень важно остановить MySQL корректно, иначе в старых журналах останутся записи, которые нужно будет применить к файлам данных! Перед повторным запуском сервера загляните в журнал ошибок MySQL. После успешного перезапуска старые журналы можно будет удалить.

Размер файла журнала и буфер журнала. Чтобы определить идеальный размер журналов, следует сопоставить накладные затраты на типичное изменение данных и время, необходимое для восстановления после аварийного останова. Если журнал слишком мал, то InnoDB придется чаще записывать контрольные точки, то есть количество записей в журнал увеличится. В предельном случае запрос на запись может быть приостановлен до завершения процесса выгрузки в файлы данных из-за того, что в журнале не осталось места. В то же время если журнал слишком велик, то InnoDB придется выполнять много работы во время восстановления после некорректного завершения работы, поэтому время данной процедуры увеличится. Однако в новых версиях MySQl этот процесс реализован намного эффективнее.

Время восстановления зависит также от объема данных и типа нагрузки. Предположим, что имеется 1 Тбайт данных и пул буферов размером 16 Гбайт, а полный размер

журнала составляет 128 Мбайт. Если количество «грязных» страниц (данные в которых модифицированы, но еще не сброшены на диск) в буферном пуле велико и они равномерно распределены по всему терабайту, то восстановление после сбоя может занять много времени. InnoDB должна будет просканировать весь журнал, проанализировать файлы данных и применить к ним необходимые изменения. Придется выполнить много операций чтения и записи! Но если изменения локализованы — скажем, часто обновляются лишь несколько сотен мегабайт, — то восстановление может пройти быстро, даже если файлы данных и журнала очень велики. Время, затраченное на этот процесс, зависит также от объема типичной модификации, который определяется средней длиной строки данных. Чем короче строки, тем больше модификаций умещается в журнал, поэтому при восстановлении InnoDB придется воспроизводить больше модификаций[1].

При модификации любых данных InnoDB помещает запись об изменении в *буфер журнала*, который хранится в памяти. InnoDB сбрасывает его в файлы журналов на диске в следующих случаях: когда буфер заполняется, когда происходит коммит транзакции или один раз в секунду — в зависимости от того, что произойдет раньше. По умолчанию размер буфера равен 1 Мбайт, а его увеличение может сократить количество операций ввода/вывода в случае больших транзакций. Размером буфера журнала управляет переменная `innodb_log_buffer_size`.

Обычно нет необходимости делать буфер очень большим. Рекомендуемый размер варьируется от 1 до 8 Мбайт, и этого более чем достаточно, если только вы не вставляете много записей с гигантскими BLOB. Записи в журнале очень компактны по сравнению с обычными данными InnoDB. Поскольку они не используют страницы как единицу хранения, то не нужно расходовать место на запись целых страниц. Кроме того, InnoDB старается сделать записи в журнале как можно более короткими. Иногда в них хранятся только номер функции на языке C и ее параметры.

Есть одно условие, при котором большой размер буфера может быть полезен: когда он помогает уменьшить конкуренцию при распределении пространства в буфере. Когда мы настраиваем серверы с большим объемом памяти, то иногда выделяем от 32 до 128 Мбайт буфера журнала просто потому, что использование такого относительно небольшого объема дополнительной памяти не наносит вреда и может помочь избежать давления на узкое место. В данном случае узким местом является конкуренция за мьютекс буфера журнала.

Следить за производительностью ввода/вывода в журнал и за его буфером позволяют раздел `LOG` в выводе, формируемом командой `SHOW INNODB STATUS`, а также переменная состояния `Innodb_os_log_written`. Хорошее эмпирическое правило таково: наблюдайте за этой переменной с интервалом 10–100 секунд и обращайте внимание на пиковые значения. На основании этой информации можно сделать вывод о том, насколько удачно выбран размер буфера журнала. Например, если

[1] Для любопытных заметим, что параметр innodb_recovery_stats Percona Server может помочь вам понять, какова рабочая нагрузка вашего сервера с точки зрения аварийного восстановления.

в пиковые периоды в журнал записывается 100 Кбайт в секунду, то буфера размером 1 Мбайт вполне достаточно.

Этот же показатель можно использовать для выбора подходящего размера файлов журнала. Если в пиковый период пишется 100 Кбайт в секунду, то журнала размером 256 Мбайт хватит для хранения записей по крайней мере за 2560 секунд, а этого, скорее всего, достаточно. Согласно эмпирическому правилу стоит делать общий размер файла журнала таким, чтобы можно было там сохранять информацию о часовой активности сервера.

Как InnoDB сбрасывает буфер журнала. Когда InnoDB сбрасывает буфер в файлы журнала на диске, она блокирует доступ к буферу с помощью мьютекса, переписывает данные на диск вплоть до нужной точки, а затем перемещает оставшиеся записи в начало буфера. Может случиться так, что к моменту освобождения мьютекса скопится несколько транзакций, готовых к записи в журнал. В InnoDB имеется механизм группового коммита, позволяющий записать их все в журнал одной операцией ввода/вывода, но он не работает в версии MySQL 5.0 при включенном двоичном журнале. В предыдущей главе мы рассказывали о групповом коммите.

Буфер журнала должен быть сброшен на устройство постоянного хранения для обеспечения долговечности зафиксированных транзакций. Если производительность для вас важнее долговечности, то можно изменить параметр `innodb_flush_log_at_trx_commit`, который контролирует, куда и как часто сбрасывается буфер журнала. Допустимы следующие значения:

- `0` — писать буфер в файл журнала и сбрасывать журнал на устройство постоянного хранения раз в секунду, но ничего не делать в момент коммита транзакции;
- `1` — писать буфер в файл журнала и сбрасывать его на устройство постоянного хранения при каждом коммите транзакции. Этот режим (самый безопасный) принимается по умолчанию, он гарантирует, что ни одна зафиксированная транзакция не будет потеряна, если только диск или операционная система не делают операцию сброса фиктивной;
- `2` — писать буфер в файл журнала при каждом коммите, но не сбрасывать его на устройство постоянного хранения. InnoDB выполняет сброс один раз в секунду. Самое важное отличие от режима `0` (из-за которого режим `2` предпочтительнее) состоит в том, что в режиме `2` транзакции не теряются в случае аварийного завершения процесса MySQL. Но если падает весь сервер или пропадает питание, потерять транзакции все-таки возможно.

Важно понимать различие между *записью* буфера в файл журнала и *сбросом* журнала на устройство постоянного хранения. В большинстве операционных систем запись буфера в журнал сводится к простому копированию данных из буфера в памяти InnoDB в кэш операционной системы, который также находится в памяти. Никакой записи на реальное устройство при этом не происходит. Поэтому режимы `0` и `2` *обычно* приводят к утрате не более чем одной секунды данных в случае сбоя или отключения питания, поскольку на протяжении этого времени информация, возможно,

существует только в кэше операционной системы. Мы говорим «обычно», потому что InnoDB старается сбрасывать файл журнала на диск примерно раз в секунду при любых обстоятельствах, но в некоторых ситуациях можно потерять транзакции более чем за одну секунду, например, когда поток сброса стопорится.

Сброс же журнала на устройство постоянного хранения означает, что InnoDB просит операционную систему действительно сбросить данные из своего кэша и убедиться в том, что они *переписаны на диск*. Это вызов, блокирующий операции ввода/вывода и не возвращающий управление, пока данные не окажутся полностью выгруженными. Поскольку запись на диск — медленная операция, то в случае, когда параметр `innodb_flush_log_at_trx_commit` равен `1`, количество транзакций, которые InnoDB способна закоммитить в секунду, может резко уменьшиться. Современные высокоскоростные накопители[1] могут записать лишь пару сотен реальных дисковых транзакций в секунду из-за ограниченных скорости вращения диска и времени позиционирования головки.

Иногда контроллер жесткого диска или операционная система подтверждают сброс, но на деле лишь копируют данные в *еще один* кэш, например в собственный кэш жесткого диска. Этот прием более быстрый, но очень опасный, поскольку данные могут быть потеряны, если внезапно пропадет питание. Такая ситуация даже хуже, чем установка параметра `innodb_flush_log_at_trx_commit` в любое значение, кроме `1`, поскольку может привести к порче данных, а не просто к потере транзакций.

Присваивание параметру `innodb_flush_log_at_trx_commit` значения, отличного от `1`, может привести к потере транзакций. Однако если долговечность (буква D (durability) в аббревиатуре ACID) для вас не очень важна, то, возможно, вы сочтете полезными и другие значения. Быть может, вас привлекают такие возможности InnoDB, как кластерные индексы, устойчивость к порче данных и блокировка на уровне строк. Не так уж редки случаи, когда InnoDB используют вместо MyISAM исключительно из соображений производительности.

Наилучшая конфигурация для высокопроизводительных транзакционных приложений получается тогда, когда `innodb_flush_log_at_trx_commit` оставляют равным `1`, а файлы журналов помещают в том RAID-массива с резервным электропитанием кэша на запись. В действительности любой сервер промышленной базы данных, который, как ожидается, будет обрабатывать большие объемы данных, должен иметь такое оборудование.

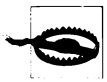
В Percona Server возможности функции `innodb_flush_log_at_trx_commit` расширены. Она сделана сеансовой переменной, а не глобальной для всего сервера. Это позволяет приложениям с различными производительностью и долговечностью использовать одну и ту же базу данных и помогает избежать одного решения для всех задач, предлагаемого стандартной MySQL.

---

[1] Мы говорим о шпиндельных накопителях с вращающимися пластинами, а не о твердотельных дисках, у которых характеристики производительности совершенно иные.

Как InnoDB открывает и сбрасывает файлы журнала и данных

Параметр `innodb_flush_method` позволяет указать, как InnoDB взаимодействует с файловой системой. Несмотря на свое название, он влияет на чтение, а не на запись данных. Значения, которые он может принимать в системах Windows и прочих, взаимно исключают друг друга: `async_unbuffered`, `unbuffered` и `normal` применимы только к Windows, другие значения в Windows не допускаются. По умолчанию на платформе Windows принимается значение `unbuffered`, а на всех остальных платформах — `fdatasync` (если команда `SHOW GLOBAL VARIABLES` показывает пустое значение этой переменной, значит, для нее действует значение по умолчанию).



При изменении режима ввода/вывода в InnoDB может существенно измениться производительность в целом, поэтому, прежде чем вносить изменения, убедитесь, что вы понимаете, что делаете!

Этот параметр может слегка сбить с толку, поскольку он влияет как на файлы журнала, так и на файлы данных, а иногда выполняет разные операции для разных типов файлов. Было бы неплохо иметь один параметр конфигурации для журналов, а другой — для файлов данных, но они объединены. Приведем возможные значения.

- `fdatasync`. Значение по умолчанию во всех системах, кроме Windows. InnoDB вызывает `fsync()` для сброса как файлов данных, так и журнала.

  Обычно InnoDB вызывает `fsync()`, а не `fdatasync()`, несмотря на то что название вроде бы свидетельствует о противоположном. Функция `fdatasync()` аналогична `fsync()`, но сбрасывает лишь данные файла, а не его метаданные (время последней модификации и т. д.). Поэтому `fsync()` выполняет больше операций ввода/вывода. Однако разработчики InnoDB, будучи очень осторожными, обратили внимание на то, что в некоторых случаях `fdatasync()` приводит к повреждению данных. Код InnoDB определяет, какие методы можно использовать безопасно, одни параметры устанавливаются на этапе компиляции, другие — на этапе выполнения. InnoDB применяет самый быстрый из безопасных методов.

  Недостаток `fsync()` заключается в том, что операционная система буферизует по крайней мере некоторые данные в собственном кэше. Теоретически это можно считать расточительной двойной буферизацией, поскольку InnoDB управляет своими буферами более разумно, чем операционная система. Но конечный эффект очень сильно зависит от операционной и файловой систем. Двойная буферизация — это не обязательно плохо, если позволяет файловой системе лучше планировать и группировать операции ввода/вывода. Некоторые файловые и операционные системы умеют накапливать процедуры записи и выполнять их одним пакетом, переупорядочивать для повышения эффективности и осуществлять вывод на несколько устройств параллельно. Кроме того, они иногда реализуют

упреждающее чтение, например просят диск заранее прочитать следующий по порядку блок, если уже поступали запросы на чтение нескольких последовательных блоков.

Иногда такие оптимизации помогают, иногда — нет. Если вам интересно, как работает ваша версия `fsync()`, можете прочитать страницу руководства fsync(2).

Параметр `innodb_file_per_table` приводит к вызову `fsync()` для каждого файла в отдельности. Поэтому записи в несколько таблиц невозможно объединить в одну операцию ввода/вывода. Таким образом, InnoDB может вынужденно увеличивать общее количество операций `fsync()`.

- `O_DIRECT`. InnoDB в зависимости от системы устанавливает флаг `O_DIRECT` или вызывает функцию `directio()` для файлов данных. Этот режим не распространяется на файлы журналов и доступен не во всех UNIX-системах. Но по крайней мере GNU/Linux, FreeBSD и Solaris (начиная с поздних выпусков версии 5.0) его поддерживают. В отличие от флага `O_DSYNC` он относится к операциям и чтения, и записи.

  В этом режиме для сброса файлов на диск также используется функция `fsync()`, но операционной системе дается указание не кэшировать данные и не прибегать к опережающему чтению. Тем самым кэш операционной системы полностью отключается и все операции чтения и записи направляются напрямую устройству хранения во избежание двойной буферизации.

  В большинстве систем это реализуется путем обращения к системному вызову `fcntl()` для установки флага `O_DIRECT` в дескрипторе файла, так что с деталями вы можете ознакомиться на странице руководства fcntl(2). В ОС Solaris данный режим подразумевает вызов функции `directio()`.

  Если упреждающее чтение осуществляется на уровне RAID-контроллера, то отменить его с помощью этого значения параметра не удастся. Он отключает упреждающее чтение только на уровне операционной или файловой системы.

  Как правило, в режиме `O_DIRECT` не следует отключать кэш записи RAID-контроллера, поскольку в типичной ситуации только это и позволяет поддерживать приемлемую производительность. Задание флага `O_DIRECT` в случае, когда между InnoDB и физическим устройством нет никакого буфера, например при отключенном кэше записи в RAID-контроллере, может привести к серьезному падению производительности. В настоящее время эта проблема менее остра из-за нескольких потоков записи (и встроенного асинхронного ввода/вывода, добавленного в MySQL 5.5).

  В этом режиме может заметно возрасти время «прогрева» сервера, особенно если кэш операционной системы очень велик. Кроме того, при небольшом буферном пуле (например, если размер по умолчанию не изменялся) сервер может работать гораздо медленнее, чем при буферизованном вводе/выводе. Это связано с тем, что операционная система не придет на выручку, сохраняя часть данных в собственном кэше. Если нужные данные отсутствуют в буферном пуле, то InnoDB будет вынуждена читать их прямо с диска.

Этот режим не влечет дополнительных накладных затрат при использовании совместно с параметром `innodb_file_per_table`. Однако может быть верным обратное: если вы не используете `innodb_file_ per_table`, то можете пострадать от некоторой сериализации ввода/вывода при использовании `O_DIRECT`. Это происходит из-за того, что в некоторых файловых системах, включая все файловые Linux-системы ext, есть по одному мьютексу на каждый индексированный дескриптор. Когда в таких файловых системах используется `O_DIRECT`, необходимо включить `innodb_file_per_table`. В следующей главе мы подробно рассмотрим файловые системы.

- `ALL_O_DIRECT`. Этот параметр доступен в Percona Server и MariaDB. Он позволяет серверу открывать файлы журнала, а не только файлы данных. Это происходит так же, как открытие файлов данных стандартной MySQL.
- `O_DSYNC`. В данном режиме при обращении к системному вызову `open()` для файлов журнала устанавливается флаг `O_SYNC`. При этом все операции записи становятся синхронными. Иными словами, функция записи не возвращает управление, пока данные не будут перенесены на диск. Указанный режим не распространяется на файлы данных.

  Разница между флагами `O_SYNC` и `O_DIRECT` заключается в том, что `O_SYNC` не отключает кэширование на уровне операционной системы. Поэтому он не устраняет двойную буферизацию и не приводит к записи прямо на диск. Если задан флаг `O_SYNC`, то операция записи сначала модифицирует данные в кэше, а потом их сохраняет.

  Хотя синхронная запись с помощью `O_SYNC` выглядит очень похоже на то, что делает функция `fsync()`, реализация может существенно отличаться как на уровне операционной системы, так и на аппаратном уровне. При использовании флага `O_SYNC` операционная система может передать флаг «применять синхронный ввод/вывод» на нижележащий аппаратный уровень, потребовав тем самым, чтобы устройство не применяло кэширование. В то же время `fsync()` говорит операционной системе, что нужно сбросить модифицированные буферы на устройство, а затем устройству посылаются команды сбросить собственные кэши (в тех случаях, когда это имеет смысл), чтобы данные гарантированно оказались записанными на физический носитель. Еще одно различие состоит в том, что при наличии флага `O_SYNC` каждый вызов `write()` или `pwrite()` синхронизирует данные и до окончания синхронизации не возвращает управление, блокируя тем самым вызывающий процесс. Напротив, запись без флага `O_SYNC` с последующим вызовом `fsync()` позволяет накапливать операции записи в кэше (при этом каждая операция завершается быстро), а потом сбрасывать их на диск пакетом.

  Опять-таки, вопреки названию, этот параметр приводит к установке флага `O_SYNC`, а не `O_DSYNC`, поскольку разработчики InnoDB обнаружили ошибки в реализации последнего. Различия во флагах `O_SYNC` и `O_DSYNC` аналогичны различиям в функциях `fsync()` и `fdatasync()`: `O_SYNC` синхронизирует данные и метаданные, а `O_DSYNC` — только данные.
- `async_unbuffered`. Этот режим подразумевается по умолчанию в Windows. В нем InnoDB для большинства операций записи использует небуферизованный ввод/

вывод с одним исключением: если параметр `innodb_flush_log_at_trx_commit` равен 2, то при записи в файлы журнала ввод/вывод буферизован.

В этом режиме InnoDB использует естественный механизм асинхронного ввода/вывода (с перекрытием) для чтения и записи при работе в ОС Windows 2000, XP и более поздних. В предыдущих редакциях Windows InnoDB применяет собственный механизм асинхронного ввода/вывода, который реализован с помощью потоков.

- `unbuffered`. Применяется только в Windows. Аналогичен режиму `async_unbuffered`, но естественный механизм асинхронного ввода/вывода не используется.
- `normal`. Применяется только в Windows. InnoDB не использует ни платформенный механизм асинхронного ввода/вывода, ни небуферизованный ввод/вывод.
- `nosync` и `littlesync`. Только для разработчиков. Эти режимы не документированы и небезопасны для применения. Их *не надо* использовать.

Табличное пространство InnoDB

InnoDB хранит данные в *табличном пространстве*, которое представляет собой некую виртуальную файловую систему, охватывающую один или несколько файлов на диске. Табличное пространство в InnoDB используется для разных целей, а не только для хранения таблиц и индексов. В нем же находятся журнал отмены (старые версии строк), буфер вставок, буфер двойной записи (описывается далее) и другие внутренние структуры.

Конфигурирование табличного пространства. Файлы, помещаемые в табличное пространство, перечисляются в конфигурационном параметре `innodb_data_file_path`. Все они будут находиться в каталоге, который задается параметром `innodb_data_home_dir`, например:

```
innodb_data_home_dir  = /var/lib/mysql/
innodb_data_file_path = ibdata1:1G;ibdata2:1G;ibdata3:1G
```

В результате создается табличное пространство размером 3 Гбайт, содержащее три файла. Иногда задают вопрос, можно ли использовать несколько файлов для распределения нагрузки на несколько накопителей, например так:

```
innodb_data_file_path = /disk1/ibdata1:1G;/disk2/ibdata2:1G;...
```

При этом файлы действительно помещаются в разные каталоги, которые в данном случае находятся на разных дисках, но InnoDB конкатенирует файлы друг за другом. Поэтому никакого реального выигрыша вы не получите. InnoDB сначала будет писать в первый файл, потом, когда он заполнится, — во второй и т. д., то есть нагрузка не распределяется так, как хотелось бы для обеспечения высокой производительности. Для этой цели лучше использовать RAID-контроллер.

Чтобы табличное пространство могло расти, когда место заканчивается, можно сделать последний файл автоматически расширяемым:

```
...ibdata3:1G:autoextend
```

По умолчанию создается один автоматически расширяемый файл размером 10 Мбайт. Если вы решите сделать файл автоматически расширяемым, стоит ограничить размер табличного пространства сверху, чтобы файл не разрастался, поскольку, достигнув определенного размера, он уже не уплотняется. Например, в следующем примере размер автоматически расширяемого файла ограничен 2 Гбайт:

```
...ibdata3:1G:autoextend:max:2G
```

Управление одним табличным пространством может вызвать сложности, особенно если оно автоматически расширяется, а вам нужно освободить место (по этой причине мы рекомендуем отключать режим автоматического расширения или по меньшей мере установить разумную верхнюю планку). В данном случае единственная возможность высвободить место — сформировать дамп данных, остановить MySQL, удалить все файлы, изменить конфигурацию, перезапустить сервер, позволить InnoDB создать новые пустые файлы и, наконец, восстановить в них данные. InnoDB очень строго относится к своему табличному пространству — вы не можете просто взять и удалить файлы или изменить их размеры. InnoDB не запустится, если обнаружит, что табличное пространство повреждено. Так же трепетно InnoDB относится к файлам журнала. Если вы привыкли без особых раздумий перемещать файлы MyISAM, будьте осторожны!

Начиная с версии MySQL 4.1, параметр `innodb_file_per_table` позволяет сконфигурировать InnoDB с одним файлом на таблицу. Данные хранятся в каталоге базы данных в файлах с именами вида `tablename.ibd`. Так проще освобождать место при удалении таблицы, к тому же этот режим полезен для распределения таблиц по разным дискам. Но если данные хранятся в нескольких файлах, больше места растрачивается впустую, поскольку вы обмениваете внутреннюю фрагментацию в одном табличном пространстве на неиспользуемое место в `.ibd`-файлах. Эта проблема больше касается маленьких таблиц, так как размер страницы в InnoDB составляет 16 Кбайт. Даже если в таблице хранится всего 1 Кбайт данных, на диске она все равно будет занимать 16 Кбайт.

Даже если режим `innodb_file_per_table` включен, главное табличное пространство все равно необходимо для файлов отмены и других системных данных. Если в нем не хранится информация, оно будет меньше размером, но тем не менее мы рекомендуем отключать автоматическое расширение, так как невозможно уплотнить файл без перезагрузки всех данных.

Некоторым режим `innodb_file_per_table` нравится просто потому, что он предоставляет дополнительные средства управления и делает структуру базы более наглядной. Например, гораздо быстрее определить размер таблицы, взглянув на соответствующий файл, нежели выполнять команду `SHOW TABLE STATUS`, которая должна выполнить некоторую работу, чтобы понять, сколько страниц выделено таблице.

Однако у `innodb_file_per_table` есть и недостаток, а именно медленное выполнение команды `DROP TABLE`. Она может выполняться настолько медленно, что способна даже вызвать заметный стопор всего сервера. Это может произойти по двум причинам.

- Удаление таблицы разрывает (удаляет) связи файла на уровне файловой системы, что в некоторых файловых системах может выполняться очень медленно (ext3,

мы смотрим на тебя). Вы можете сократить продолжительность этой операции с помощью трюков в файловой системе: связать файл `.ibd` с файлом нулевого размера, а затем удалить файл вручную, а не ждать, пока это сделает MySQL.

- Когда вы включаете этот параметр, каждая таблица получает собственное табличное пространство внутри InnoDB. Оказывается, что удаление табличного пространства фактически требует от InnoDB блокировки и сканирования буферного пула, в то время как он ищет страницы, принадлежащие этому табличному пространству, что очень медленно выполняется на сервере с большим буферным пулом. Если вы собираетесь удалять множество таблиц InnoDB, включая временные таблицы, и вы используете `innodb_file_per_table`, то можете воспользоваться исправлением, включенным в Percona Server. Оно позволяет серверу лениво инвалидировать страницы, принадлежащие удаленным таблицам. Для этого нужно просто установить параметр `innodb_lazy_drop_table`.

Какова же итоговая рекомендация? Мы советуем использовать `innodb_file_per_table` и, чтобы упростить себе жизнь, ограничить размер разделенного табличного пространства. Если даже в этом случае вы столкнетесь с проблемами, рассмотрите одно из исправлений, которые мы предложили.

Следует также отметить, что файлы InnoDB не обязательно хранить в традиционной файловой системе. Как и многие другие серверы базы данных, InnoDB позволяет размещать их в неформатированном разделе диска. Однако современные файловые системы умеют эффективно работать с довольно большими объектами, поэтому в таком режиме нет особого смысла. Использование неформатированных устройств может дать прирост производительности на несколько процентов, но мы считаем, что такой мизерный выигрыш не оправдывает неудобства, связанные с невозможностью манипулировать данными как файлами. Если они находятся на неформатированном устройстве, то о командах `mv`, `cp` и прочих можно забыть. В общем, из-за того крохотного выигрыша, который дают неформатированные устройства, не стоит суетиться.

Старые версии строк и табличное пространство. В условиях интенсивной записи табличное пространство InnoDB может стать очень большим. Если транзакция долго остается открытой (даже не делая ничего полезного) и при этом установлен уровень изоляции `REPEATABLE READ`, то InnoDB не может удалить старые версии строк, поскольку они должны быть видны незакоммиченным транзакциям. InnoDB хранит старые версии в табличном пространстве, поэтому по мере обновления данных оно продолжает расти. Иногда проблема связана не с открытыми транзакциями, а просто с характером рабочей нагрузки: чисткой занимается только один поток (так было в старых версиях MySQL), он может просто не успевать удалять старые версии строк.

В любом случае команда `SHOW INNODB STATUS` поможет идентифицировать причину проблемы. Взгляните на длину списка историй — она покажет размер журнала отмены в страницах.

Этому можно найти подтверждение, если взглянуть на первую и вторую строки в разделе `TRANSACTIONS`, там показаны текущий номер транзакции и точка, до которой

дошла чистка. Если разница велика, значит, скопилось много невычищеных транзакций. Приведем пример:

```
------------
TRANSACTIONS
------------
Trx id counter 0 80157601
Purge done for trx's n:o <0 80154573 undo n:o <0 0
```

Идентификатор транзакции — это 64-разрядное число, составленное из двух 32-разрядных (в новых версиях InnoDB это шестнадцатеричное число), поэтому для вычисления разности придется немного посчитать. В данном случае все просто: так как старшие разряды равны 0, то мы имеем 80 157 601 – 80 154 573 = 3028 потенциально невычищеных транзакций (утилита innotop проделает за вас все вычисления). Мы говорим «потенциально», поскольку большая разность еще не означает, что невычищеных строк много. Старые версии строк создаются только для транзакций, которые изменяют данные, а может существовать множество транзакций, в которых данные не изменялись (и наоборот, одна транзакция может изменить много строк).

Если невычищеных транзакций много и по этой причине табличное пространство продолжает расти, то можно принудительно замедлить работу MySQL, чтобы поток очистки InnoDB справлялся с нагрузкой. Звучит не слишком привлекательно, но альтернативы нет. В противном случае InnoDB будет продолжать писать данные и заполнять диск, пока на нем не кончится место или не будет достигнута заданная верхняя граница табличного пространства.

Чтобы притормозить запись, присвойте переменной `innodb_max_purge_lag` значение, отличное от 0. Оно равно максимальному количеству транзакций, ожидающих очистки; по достижении этого порога InnoDB начинает задерживать выполнение запросов на обновление данных. Чтобы выбрать подходящее значение, нужно знать конкретную рабочую нагрузку. Например, если типичная транзакция изменяет в среднем 1 Кбайт данных и вы готовы смириться с 100 Мбайт невычищеных строк в табличном пространстве, то можете задать значение `100000`.

Имейте в виду, что наличие невычищеных версий строк отражается на всех запросах, поскольку из-за них увеличивается размер таблиц и индексов. Если поток очистки не справляется с нагрузкой, то производительность может заметно упасть. Задание параметра `innodb_max_purge_lag` тоже снижает скорость выполнения запросов, но это меньшее из зол[1].

В новых версиях MySQL и даже более старых версиях Percona Server и MariaDB процесс очистки значительно улучшен и отделен от других внутренних вспомогательных задач. Можно даже создать несколько выделенных потоков очистки для того, чтобы выполнить эту работу быстрее. Этот вариант, если он вам доступен, намного лучше, чем притормаживание сервера.

---

[1] Отметим, что способ, которым этот прием должен быть реализован, остается предметом споров (см. руководство по MySQL, ошибка 60776 для деталей).

Буфер двойной записи

В InnoDB *буфер двойной записи* применяется для того, чтобы избежать повреждения данных в случае частичной записи страницы. Частичная запись страницы возникает, когда операция записи на диск выполняется не полностью, так что записанной оказывается только часть страницы размером 16 Кбайт. Причин для этого много — сбои, ошибки и т. д. В данном случае буфер двойной записи предотвращает повреждение данных.

Буфер двойной записи представляет собой зарезервированную область в табличном пространстве, достаточно большую для хранения 100 страниц в одном непрерывном блоке. По существу, это резервная копия недавно записанных страниц. Когда InnoDB сбрасывает страницы из буферного пула на диск, она записывает (и сбрасывает) их сначала в буфер двойной записи и лишь потом — в ту основную область данных, где им и место. Тем самым гарантируется, что каждая операция записи страницы неделима и долговечна.

Означает ли это, что каждая страница записывается дважды? Да, означает, но поскольку InnoDB пишет в буфер двойной записи сразу несколько страниц и лишь потом вызывает `fsync()` для синхронизации с диском, то на производительности это почти не отражается — обычно потери составляют не более нескольких процентов, а не увеличиваются в два раза, хотя накладные расходы более заметны на твердотельных дисках, что мы обсудим в следующей главе. Важнее тот факт, что эта стратегия позволяет гораздо эффективнее использовать файлы журнала. Поскольку буфер двойной записи дает InnoDB надежную гарантию того, что данные не повреждены, то в журнал помещаются не страницы целиком, а скорее двоичные разности.

Если при сохранении страницы произойдет частичная запись в сам буфер двойной записи, то оригинальная страница все равно окажется на диске в том месте, где ей и положено быть. На этапе восстановления InnoDB возьмет оригинальную страницу вместо ее поврежденной копии в буфере двойной записи. Если же запись в буфер завершилась успешно, а в саму таблицу — с ошибкой, то во время восстановления InnoDB воспользуется копией из буфера двойной записи. InnoDB понимает, что данные повреждены, так как в конце каждой страницы имеется контрольная сумма (она записывается последней), — если эта сумма не соответствует содержимому страницы, значит, страница повреждена. Поэтому в ходе восстановления InnoDB читает каждую страницу, находящуюся в буфере двойной записи, и проверяет контрольную сумму. Если она неверна, то считывается оригинальная страница.

В некоторых случаях без буфера двойной записи можно обойтись — например, его можно отключить на подчиненных серверах. Кроме того, некоторые файловые системы (в частности, ZFS) делают то же самое самостоятельно, поэтому излишне повторять процедуру на уровне InnoDB. Чтобы отключить буфер двойной записи, присвойте параметру `innodb_doublewrite` значение `0`. В Percona Server можно настроить буфер двойной записи так, чтобы он хранился в собственном файле. Тем самым можно отделить эту рабочую нагрузку от остальной части работы сервера, разместив их на отдельных дисках.

Прочие параметры настройки ввода/вывода

Параметр `sync_binlog` управляет тем, как MySQL сбрасывает двоичный журнал на диск. По умолчанию он равен `0`, то есть MySQL вообще не выполняет сброс, оставляя его на усмотрение операционной системы. Значение, превышающее `0`, интерпретируется как количество операций записи в двоичный журнал между двумя последовательными сбросами (операцией считается одна команда, если режим autocommit включен, и одна транзакция — в противном случае). Обычно этот параметр устанавливают в `0` или `1`.

Если `sync_binlog` отличен от `1`, то в случае сбоя двоичный журнал может оказаться не синхронизированным с транзакционными данными. Это вполне способно привести к сбою в репликации и сделать невозможным восстановление на определенный момент в прошлом. Однако уровень безопасности, получаемый установкой этого параметра в `1`, требует весьма высоких затрат. Для синхронизации двоичного журнала с журналом транзакций MySQL должна сбрасывать два файла в двух разных местах. Для этого может потребоваться относительно медленная операция подвода головки к дорожке.

Как и в случае с файлом журнала InnoDB, помещение двоичного журнала в том RAID-массива, оборудованного кэшем записи с резервным питанием, нередко дает огромный прирост производительности. Запись и сброс двоичных журналов — более затратные операции, чем запись и сброс журналов транзакций InnoDB, потому что, в отличие от журналов транзакций InnoDB, каждая запись в двоичные журналы увеличивает их размер. Для каждой записи требуется обновление метаданных на уровне файловой системы. Таким образом, установка `sync_binlog = 1` может намного сильнее ухудшить производительность, чем установка `innodb_flush_log_at_trx_commit = 1`, особенно в сетевых файловых системах, таких как NFS.

Замечание о двоичном журнале, не относящееся к производительности: если вы собираетесь воспользоваться параметром `expire_logs_days` для автоматического удаления старых двоичных журналов, то не уничтожайте их вручную командой `rm`. В этом случае сервер запутается и откажется выполнять автоматическое удаление, а команда `PURGE MASTER LOGS` перестанет работать. Если вы окажетесь в такой ситуации, то нужно будет вручную синхронизировать файл `hostname-bin.index` со списком файлов на диске.

Более подробно обсуждать RAID-массивы будем в главе 9, а сейчас стоит отметить, что высококачественные RAID-контроллеры, оборудованные кэшем с резервным батарейным питанием и работающие в режиме отложенной записи, могут справляться с *тысячами* операций в секунду и при этом обеспечивать надежное хранение. Поскольку кэш запитан от батареи, то его содержимое не утрачивается даже при отключении питания. После его восстановления RAID-контроллер переписывает данные из кэша на диск еще до того, как сделать диск доступным для работы. Таким образом, хороший RAID-контроллер, оборудованный кэшем записи, с резервным питанием может резко повысить производительность и потому является отличным капиталовложением. Конечно же, твердотельный накопитель — еще один вариант, мы также расскажем об этом в следующей главе.

MyISAM

Начнем с рассмотрения того, как MyISAM выполняет ввод/вывод для индексов. Обычно изменение индекса сбрасывается на диск после каждой записи. Но если вы производите много модификаций в таблице, то будет быстрее сгруппировать операции записи вместе.

Один из способов добиться этого — воспользоваться командой `LOCK TABLES`, которая откладывает запись до момента разблокировки таблиц. Этот прием способствует повышению производительности, так как позволяет точно управлять тем, какие операции записи откладываются и когда происходит сброс на диск. Можно отложить запись именно для интересующих вас команд.

Запись в индекс можно отложить также с помощью переменной `delay_key_write`. В этом случае блоки из буфера ключей не сбрасываются на диск до момента закрытия таблицы[1]. Допустимы следующие значения:

- `OFF` — MyISAM сбрасывает измененные блоки из буфера ключей после каждой записи, если только таблица не блокирована командой `LOCK TABLES`;
- `ON` — включен режим отложенной записи ключей, но только для таблиц, созданных с параметром `DELAY_KEY_WRITE`;
- `ALL` — для всех таблиц типа MyISAM используется отложенная запись ключей

В некоторых случаях отложенная запись оказывается кстати, но обычно этот прием не приводит к резкому увеличению производительности. Наиболее полезна она в ситуации, когда размер данных мал, коэффициент попадания в кэш при чтении высокий, а при записи — низкий. К тому же у этого режима есть ряд недостатков.

- Если сервер завершает работу аварийно, а блоки не сброшены на диск, то индекс будет испорчен.
- Если было отложено много операций записи, то MySQL потратит больше времени на закрытие таблицы, поскольку вынуждена ждать завершения записи буферов на диск. В версии MySQL 5.0 это приводит к длительным блокировкам доступа к кэшу таблиц.
- По тем же причинам выполнение команды `FLUSH TABLES` может занимать много времени. А это, в свою очередь, может увеличить время выполнения команды `FLUSH TABLES WITH READ LOCK` при создании мгновенного снимка для менеджера логических томов (LVM), а также других операций резервного копирования.
- Несброшенные «грязные» блоки в буфере ключей могут не оставить места для новых блоков, считываемых с диска. В таком случае выполнение запроса будет приостановлено до тех пор, пока MyISAM не освободит достаточно места в буфере ключей.

[1] Таблица может быть закрыта по нескольким причинам. Например, сервер может закрыть таблицу, поскольку в кэше таблиц не осталось места или кто-то другой выполнил команду FLUSH TABLES.

Помимо настройки ввода/вывода для индексов MyISAM, можно сконфигурировать метод восстановления после того, как данные были испорчены. Параметр `myisam_recover` управляет алгоритмом поиска и исправления ошибок. Его можно задавать как в конфигурационном файле, так и в командной строке. Чтобы просмотреть (но не изменить) его, воспользуйтесь следующей SQL-командой (здесь нет опечатки — системная переменная действительно называется не так, как параметр в командной строке):

```
mysql> SHOW VARIABLES LIKE 'myisam_recover_options';
```

Если этот режим включен, то MySQL проверяет, не испорчены ли таблицы в момент открытия, и при обнаружении ошибок тут же исправляет их. Параметр может принимать следующие значения:

- `DEFAULT` (или не задан) — MySQL попытается исправить таблицу, помеченную как сбойная, или не помеченную как корректно закрытая. Это режим, заданный по умолчанию, в котором никаких других действий при восстановлении не предпринимается. Значение `DEFAULT`, в отличие от большинства других переменных, не означает, что нужно восстановить режим, указанный на этапе компиляции, а интерпретируется просто как «не задано»;
- `BACKUP` — заставляет MySQL записать резервную копию файла данных в файл с расширением `BAK`, который позже можно будет исследовать;
- `FORCE` — требует продолжить восстановление, даже если в `.MYD`-файле потеряно более одной строки;
- `QUICK` — пропускает фазу восстановления, если отсутствуют блоки удаления. Так называются блоки, занятые удаленными строками. Они все еще занимают место в файле, но могут быть повторно использованы во время выполнения команды `INSERT`. Этот режим может оказаться полезным, так как восстановление большой MyISAM-таблицы иногда занимает весьма длительное время.

Можно использовать несколько значений, указываемых через запятую. Например, комбинация `BACKUP, FORCE` принудительно продолжит восстановление и создаст резервную копию.

Мы рекомендуем включать этот параметр, особенно если имеется всего несколько небольших таблиц MyISAM. Запускать сервер с испорченными таблицами MyISAM опасно, поскольку иногда это приводит к дальнейшему повреждению данных и даже аварийному останову сервера. Однако при наличии больших таблиц автоматическое восстановление может оказаться непрактичным: сервер должен проверить и исправить любую таблицу MyISAM в момент ее открытия, что, конечно, неэффективно. MySQL обычно приостанавливает на это время работу всех соединений. Если количество таблиц MyISAM велико, то, пожалуй, стоит воспользоваться менее навязчивой процедурой: выполнить команды `CHECK TABLES` и `REPAIR TABLES` после запуска сервера[1]. В любом случае проверка и исправление таблиц очень важны.

[1] Некоторые системы Debian делают это автоматически, таким образом, маятник уходит слишком далеко в другую сторону. Не рекомендуется настраивать такие действия в качестве поведения по умолчанию, как сделано в Debian, — это решение должен принять администратор базы данных.

Еще одна полезная настройка MyISAM — доступ к файлам данных в режиме проецирования в память. При этом MyISAM обращается к `.MYD`-файлам непосредственно через кэш страниц операционной системы, обходясь без затратных системных вызовов. Начиная с версии MySQL 5.1, включить режим проецирования в память можно с помощью параметра `myisam_use_mmap`. В более старых версиях проецирование в память разрешено только для сжатых MyISAM-таблиц.

Настройка конкурентного доступа в MySQL

Если MySQL работает в режиме высокого конкурентного доступа, то могут возникнуть узкие места, не встречающиеся при других условиях. В этом разделе мы расскажем, как распознать подобные проблемы и добиться наилучшей производительности при такой рабочей нагрузке в подсистемах хранения MyISAM и InnoDB.

InnoDB

Подсистема InnoDB изначально проектировалась для работы в условиях высокого конкурентного доступа, но она неидеальна. До сих пор видно, что своими корнями архитектура InnoDB уходит во времена систем с ограниченной памятью, одним процессором и одним диском. Некоторые характеристики InnoDB, относящиеся к производительности, резко ухудшаются в условиях высокой конкуренции, и тогда остается единственный выход — ограничить конкурентный доступ. Для диагностики проблем с ним можно использовать методику, описанную в главе 3.

Когда вы сталкиваетесь с проблемами, вызванными конкурентным доступом в InnoDB, очевидное решение — обновить сервер. По сравнению с текущими более старые версии, такие как MySQL 5.0 и ранняя MySQL 5.1, при высоком уровне конкурентного доступа были непревзойденными источниками проблем. Все процессы выстраивались в очереди к глобальным мьютексам, например к мьютексу буферного пула, и сервер практически останавливался. Если вы перешли на одну из новых версий MySQL, в большинстве случаев вам не придется ограничивать конкурентный доступ.

В InnoDB имеется собственный планировщик потоков, который управляет тем, как потоки входят в ядро для доступа к данным, и тем, что они могут делать, находясь в ядре. Самый простой способ ограничить степень параллелизма — воспользоваться переменной `innodb_thread_concurrency`, которая определяет, сколько потоков могут находиться в ядре одновременно. Значение `0` символизирует отсутствие ограничения на количество потоков. При возникновении проблем с конкурентным доступом в InnoDB на эту переменную следует обратить внимание в первую очередь[1].

[1] Фактически при некоторых нагрузках система, реализующая ограничения конкуренции, сама может стать узким местом, поэтому в одних случаях ее нужно включить, а в других — отключить. Что именно делать, покажет профилирование.

Невозможно заранее сказать, какое значение лучше всего подходит для заданной архитектуры и существующей рабочей нагрузки. Теоретически можно воспользоваться следующей формулой:

```
concurrency = Количество ЦП * Количество дисков * 2
```

Однако на практике может оказаться, что лучше устанавливать гораздо меньшее значение. Чтобы подобрать походящее значение этого параметра для своей системы, придется поэкспериментировать.

Если в ядре уже находится больше потоков, чем разрешено, то никакой другой поток не сможет в него войти. В InnoDB применяется двухшаговая процедура, смысл которой заключается в том, чтобы сделать вход в ядро максимально эффективным. Эта политика сокращает накладные затраты на контекстные переключения, свойственные планировщику операционной системы. Поток сначала засыпает на `innodb_thread_sleep_delay` микросекунд, а потом вновь пытается войти. Если это по-прежнему не получается, то поток становится в очередь ожидания и уступает управление операционной системе.

По умолчанию на первом шаге поток спит в течение 10 000 микросекунд. При высокой конкуренции, когда процессор не используется полностью, потому что множество потоков находится в состоянии «спит перед постановкой в очередь», можно попробовать изменить этот параметр. Принимаемое по умолчанию значение может оказаться слишком велико, если имеется множество мелких запросов, поскольку к времени обработки каждого запроса добавляется 10 микросекунд.

Если поток уже вошел в ядро, то он получает определенное количество «билетов», позволяющих ему вернуться в ядро «бесплатно», то есть без проверки условий конкуренции. Тем самым налагаются ограничения на объем работы, который поток может выполнить перед тем, как встать в очередь наравне с остальными ожидающими потоками. Переменная `innodb_concurrency_tickets` определяет количество «билетов». Изменять ее приходится редко — разве что в случае, когда имеется большое количество долго выполняющихся запросов. «Билеты» выдаются на запрос, а не на транзакцию. После того как запрос выполнен, все оставшиеся «билеты» аннулируются.

Кроме узких мест, возникающих из-за буферного пула и других структур, существует узкое место и на этапе коммита транзакции. Оно связано главным образом с вводом/выводом и вызвано операциями сброса. Переменная `innodb_commit_concurrency` определяет, сколько потоков могут одновременно выполнять коммит. Ее имеет смысл настраивать, если из-за слишком низкого значения переменной `innodb_thread_concurrency` потоки начинают пробуксовывать.

Наконец, есть новое, возможно, достойное решение — использование пула потоков для ограничения конкурентного доступа. Исходная реализация пула потоков находилась в заброшенном дереве исходников MySQL 6.0 и имела серьезные недостатки. Но этот прием был заново реализован в MariaDB, а Oracle недавно выпустила коммерческий плагин, обеспечивающий пул потоков для MySQL 5.5. У нас нет богатого опыта использования этих решений, поэтому мы запутаем вас еще больше и скажем, что ни одна из этих реализаций, похоже, не удовлетворяет Facebook, который решает

свои уникальные задачи с помощью так называемых возможностей контроля доступа в собственной закрытой ветви MySQL. Надеемся, к четвертому изданию этой книги мы накопим больше знаний и сможем вам рассказать, что такое пулы потоков и при каких условиях они работают.

MyISAM

При определенных условиях MyISAM допускает конкурентные вставку и чтение и позволяет планировать некоторые операции, чтобы по возможности уменьшить количество блокировок.

Но прежде, чем мы приступим к описанию параметров настройки конкурентного доступа, важно понять, как MyISAM удаляет и вставляет данные. Операция удаления не реорганизует всю таблицу, а лишь помечает строки соответствующим признаком, оставляя дыры в таблице. MyISAM старается по возможности заполнить эти дыры, используя освободившееся место для вставки новых строк. Если дыр нет, то следующие данные дописываются в конец таблицы.

Несмотря на то что в подсистеме MyISAM есть табличные блокировки, она может дописывать новые строки одновременно с чтением. Для этого сервер следит, чтобы операции чтения останавливались на последней строке, которая существовала к моменту их начала. Таким образом удается избежать несогласованных операций.

Однако обеспечить согласованность чтения, когда что-то изменяется в середине таблицы, гораздо труднее. Популярный способ решения этой проблемы дает технология MVCC: она позволяет считывать старые версии данных, пока писатели создают новые версии. Однако MyISAM, в отличие от InnoDB, не поддерживает MVCC, поэтому не поддерживает и конкурентную вставку — только вставку в конец таблицы.

Поведение, характерное для конкурентной вставки, в MyISAM можно сконфигурировать с помощью переменной `concurrent_insert`, которая принимает следующие значения:

- `0` — MyISAM вообще не допускает конкурентных вставок; любая вставка монопольно блокирует таблицу;
- `1` — это значение по умолчанию. MyISAM допускает конкурентную вставку, если в таблице нет дыр;
- `2` — это значение доступно для MySQL 5.0 и более новых версий. В данном случае конкурентная вставка принудительно производится в конец таблицы, даже если в ней есть дыры. Если же ни один поток не читает из таблицы, то MySQL помещает новые строки в дыры. В таком режиме может возрасти фрагментация данных.

Можно также сконфигурировать MySQL так, чтобы некоторые операции откладывались на более позднее время, когда для большей эффективности их можно будет сгруппировать. Например, переменная `delay_key_write` позволяет задать режим отложенной записи в индекс, о чем мы уже упоминали в текущей главе. Тут возникает

знакомый компромисс: записывать в индекс немедленно (безопасно, но затратно) или подождать в надежде, что до момента записи не произойдет сбоя электропитания (быстрее, но любая неисправность может привести к серьезному повреждению индекса из-за его неактуальности).

Параметр `low_priority_updates` позволяет назначать командам `INSERT`, `REPLACE`, `DELETE` и `UPDATE` более низкий приоритет, чем команде `SELECT`. Этот режим эквивалентен применению подсказки `LOW_PRIORITY` ко всем запросам обновления. Это действительно очень важный параметр при использовании MyISAM — он позволяет добиться хорошего уровня конкуренции для запросов `SELECT`, которые в противном случае могли бы зависать из-за очень небольшого числа запросов на запись, получающих высокий приоритет.

Наконец, хотя проблемы масштабируемости чаще обсуждаются в контексте InnoDB, у подсистемы MyISAM тоже в течение длительного времени возникали сложности с мьютексами. В версии MySQL 4.0 и более ранних любой доступ к буферу ключей был защищен глобальным мьютексом, что ухудшало масштабируемость в системах с несколькими процессорами и дисками. В версии MySQL 4.1 код работы с буфером ключей был усовершенствован, и теперь этой проблемы нет, тем не менее каждый буфер ключей по-прежнему защищен мьютексом. Это вызывает сложности, когда поток копирует блоки ключей из буфера в собственную локальную память, а не читает их с диска. Диск перестает быть узким местом, но теперь таковым является доступ к данным в буфере ключей. Иногда эту проблему удается решить за счет организации нескольких буферов ключей, но подобный подход не всегда помогает. Например, ничего нельзя сделать, если все упирается в единственный индекс. В результате конкурентные запросы `SELECT` могут выполняться на машинах с несколькими процессорами гораздо медленнее, чем на машине с одним процессором, даже в тех случаях, когда кроме них вообще ничего не выполняется. MariaDB предлагает сегментированные (секционированные) буферы ключей, которые могут значительно помочь при возникновении этой проблемы.

Настройка с учетом рабочей нагрузки

Конечная цель настройки сервера — адаптировать его к конкретной рабочей нагрузке. Для этого нужно хорошо знать все виды выполняющихся задач: их количество, типы и частоту. Причем речь идет не только о запросах, но и о таких операциях, как подключение к серверу и сброс таблиц.

Первое, что нужно сделать, если вы этого еще не сделали, — познакомиться со своим сервером. Узнать, какие запросы он выполняет. Последить за ним с помощью утилиты, такой как innotop, а также использовать pt-query-digest для создания отчета по запросу. Полезно знать не только что он делает в общем, но и на что тратит основное время каждый запрос. В главе 3 рассказывается, как это выяснить.

Попробуйте журналировать все запросы в периоды времени, когда сервер работает на полную мощность, поскольку это лучший способ понять, какие виды запросов

страдают больше всего. В то же время фиксируйте снимки списка процессов и агрегируйте их по состоянию или команде (сделать это может innotop, или вы можете использовать скрипты, показанные в главе 3). Например, многие ли из них копируют результаты во временную таблицу или занимаются сортировкой результатов? Если да, значит, нужно обратить внимание на конфигурационные параметры, относящиеся к временным таблицам и буферам сортировки (возможно, понадобится оптимизировать и сами запросы).

Оптимизация работы с полями типа BLOB и TEXT

Столбцы типа `BLOB` и `TEXT` представляют для MySQL особый вид рабочей нагрузки. (Для простоты будем называть те и другие просто `BLOB`, потому что они принадлежат к одному и тому же классу типов данных.) `BLOB` имеют ряд ограничений, из-за которых сервер работает с ними несколько иначе, чем с другими типами. В частности, сервер не может использовать для `BLOB` временные таблицы в памяти[1]. Следовательно, если запросу, включающему `BLOB`, требуется временная таблица, то она, независимо от величины, создается на диске. Это крайне неэффективно, особенно если в остальном запрос маленький и мог бы быть выполнен очень быстро. На работу с временной таблицей может затрачиваться львиная доля времени обработки запроса.

Уменьшить эти затраты можно двумя способами: преобразовать значение к типу `VARCHAR` с помощью функции `SUBSTRING()` (см. главу 4) или ускорить работу с временными таблицами.

Последнее проще всего сделать, поместив временные таблицы в файловую систему, находящуюся целиком в памяти (tmpfs в случае GNU/Linux). Это устраняет часть накладных затрат, хотя все равно работает гораздо медленнее, чем временные таблицы. Использование файловой системы в памяти помогает, поскольку операционная система стремится избежать записи на диск[2]. Обычные файловые системы тоже поддерживают кэш в памяти, но операционная система может сбрасывать кэш на диск каждые несколько секунд. В файловой системе tmpfs сбросы не производятся. К тому же при проектировании файловой системы tmpfs специально были поставлены две цели: обеспечение низких накладных затрат и простота. В частности, для этой файловой системы не выполняются действия, способствующие ее восстановлению. Это ускоряет работу.

Где именно создаются временные таблицы, определяет параметр `tmpdir`. Следите за тем, чтобы при заполнении файловой системы в ней всегда было достаточно места для временных таблиц. При необходимости можно задать даже несколько мест для их размещения, MySQL будет использовать временные таблицы по кругу.

Если `BLOB` очень велики, а вы работаете с InnoDB, то, возможно, придется увеличить размер буфера журнала. Мы уже подробно обсуждали этот вопрос в настоящей главе.

---

[1] Последние версии Percona Server в некоторых случаях снимают эти ограничения.

[2] Данные все равно могут записываться на диск, если ОС вынуждена прибегать к подкачке.

Для длинных столбцов переменной длины (например, типа `BLOB`, `TEXT` и больших столбцов с символьными данными) InnoDB хранит префикс длиной 768 байт в самой странице вместе со строкой[1]. Если значение в столбце длиннее префикса, то для хранения остатка InnoDB может выделить внешнюю память вне строки. Эта память сегментируется страницами длиной 16 Кбайт — такими же, как и все остальные страницы в InnoDB, причем каждому столбцу отводится отдельная страница (столбцы не используют совместно внешнюю память). InnoDB выделяет столбцу внешнюю память по одной странице за раз до тех пор, пока не будут выделены 32 страницы. После этого за один раз выделяются сразу 64 страницы.

Отметим: мы сказали, что InnoDB *может* выделить внешнюю память. Если полная длина строки, включая все значение длинного столбца, меньше максимально допустимой в InnoDB длины строки (чуть меньше 8 Кбайт), то InnoDB не станет выделять внешнюю память, даже если длина значения в длинном столбце превышает длину префикса.

Наконец, отметим, что когда InnoDB обновляет длинный столбец, для которого выделена внешняя память, то значение в данном местоположении не изменяется. Вместо этого очередное значение записывается на новое место во внешней памяти, а старое значение удаляется.

Все это приводит к таким последствиям.

- На хранение длинных столбцов в InnoDB может впустую расходоваться много места. Например, если для размещения значения в строке не хватает всего лишь 1 байта, то для этого единственного не поместившегося байта будет выделена целая страница и большая ее часть окажется потраченной впустую. Аналогично если значение лишь чуть-чуть длиннее 32 страниц, то для его хранения *могут* фактически использоваться 96 страниц на диске.
- Наличие внешней памяти отключает адаптивный хеш-индекс, поскольку его использование подразумевает сравнение полных длин столбцов, чтобы гарантировать, что найдены нужные данные. (Хеш позволяет InnoDB очень быстро находить «примерно то, что нужно», но затем надо проверить, что догадка верна.) Так как адаптивный хеш-индекс целиком находится в памяти и строится поверх страниц в буферном пуле, к которым часто производятся обращения, то с внешней памятью он работать не может.
- Из-за длинных значений могут медленно выполняться запросы, содержащие условие `WHERE`, для которого нет подходящего индекса. Перед проверкой `WHERE` MySQL читает все запрошенные столбцы, поэтому может потребоваться, чтобы InnoDB прочитала все внешние страницы. Затем проверяется выполнение условия `WHERE` и все прочитанные данные отбрасываются. Никогда не следует вы-

[1] Этого вполне достаточно для создания по столбцу индекса с ключом длиной 255 символов даже в кодировке UTF-8, где один символ может кодироваться 3 байтами. Этот префикс относится к файловому формату Antelope InnoDB. он не применяется к формату Barracuda, который доступен в MySQL 5.1 и новее (хотя по умолчанию не включен).

бирать ненужные столбцы, но в данном случае особенно важно не делать этого. Если обнаруживается, что это ограничение относится к вашим запросам, можно попробовать использовать покрывающие индексы.

- Если в одной таблице много длинных столбцов, то, возможно, будет лучше объединить их в один столбец, может быть, в виде XML-документа. Тогда для объединенного значения будет выделена только одна область во внешней памяти, а не по отдельному набору страниц на каждый столбец.
- Иногда можно значительно увеличить свободное пространство и улучшить производительность, если поместить длинные столбцы в `BLOB` и сжать функцией `COMPRESS()` либо сжимать данные на уровне приложения перед отправкой MySQL.

Оптимизация файловой сортировки

Вспомним: в главе 6 говорилось, что MySQL использует два алгоритма файловой сортировки. Двухпроходный алгоритм применяется, если суммарная длина всех столбцов, отбираемых запросом, плюс длина столбцов, упоминаемых в разделе `ORDER BY`, превышает `max_length_for_sort_data` байт. MySQL также использует этот алгоритм тогда, когда хотя бы один из запрошенных столбцов — пусть даже он не встречается в `ORDER BY` — имеет тип `BLOB` или `TEXT`. (Вы можете использовать функцию `SUBSTRING()` для преобразования таких столбцов в тип, к которому применим однопроходный алгоритм.)

MySQL использует две переменные, которые помогают управлять файловой сортировкой. На выбор алгоритма можно повлиять, задав параметр `max_length_for_sort_data`[1]. Поскольку в однопроходном алгоритме для каждой строки создается буфер фиксированного размера, то при подборе значения `max_length_for_sort_data` учитывают максимальную, а не фактическую длину столбцов типа `VARCHAR`. Это одна из причин, почему мы рекомендуем не задавать для таких столбцов бóльшую длину, чем необходимо.

При сортировке по столбцам типа `BLOB` или `TEXT` MySQL принимает во внимание только префикс, а остаток значения игнорирует. Связано это с тем, что для значений нужно выделить структуру фиксированной длины, а затем скопировать в нее префикс из внешней памяти. Длина такого префикса задается переменной `max_sort_length`.

К сожалению, MySQL не дает никакой информации о выбранном алгоритме сортировки. Если после увеличения переменной `max_length_for_sort_data` интенсивность использования диска возрастает, потребление центрального процессора падает, а переменная состояния `Sort_merge_passes` начинает расти быстрее, чем раньше, значит, скорее всего, увеличилось число сортировок однопроходным алгоритмом.

---

[1] MySQL 5.6 внесет изменения в способ использования буфера сортировки в запросах с ключевым словом LIMIT и устранит проблему, из-за которой большой буфер сортировки использовался для выполнения затратной процедуры настройки. Поэтому при обновлении до MySQL 5.6 вы должны тщательно проверить все сделанные вами настройки.

Завершение базовой конфигурации

Мы закончили экскурсию по внутреннему устройству сервера — надеемся, она вам понравилась! Теперь вернемся к примерному файлу конфигурации и посмотрим, как выбрать значения для оставшихся настроек.

Мы уже рассказали, как выбрать значения для общих параметров, таких как каталог данных, кэши InnoDB и MyISAM, журналы и несколько других настроек. Давайте рассмотрим, что осталось.

- `tmp_table_size` и `max_heap_table_size`. Эти параметры определяют, как может увеличиться временная таблица в памяти, с помощью подсистемы хранения Memory. Если размер неявной временной таблицы превышает один из этих параметров, она будет преобразована в таблицу MyISAM на диске и сможет продолжать расти. (Неявная временная таблица — это та, которую создаете не вы — ее создает сервер для хранения промежуточных результатов при выполнении запроса.)

  Вы должны просто присвоить обеим переменным одно и то же значение. Для образца файла конфигурации мы выбрали значение `32M`. Этого может быть недостаточно, но остерегайтесь задавать этой переменной слишком большое значение. Временным таблицам целесообразно находиться в памяти, но лишь до тех пор, пока они не стали просто огромными. В этом случае следует использовать таблицы на диске, иначе у сервера может закончиться память.

  Если предположить, что ваши запросы не создают огромных временных таблиц (чего, как правило, можно избежать с помощью правильного индексирования и проектирования запросов), то стоит присвоить этим переменным достаточно большие значения, чтобы серверу не пришлось преобразовывать таблицы в памяти в таблицы на диске. Эта процедура будет видна в списке процессов.

  Для того чтобы узнать, как часто создаются временные таблицы и попадают ли они на диск, посмотрите, как изменяются с течением времени счетчики `SHOW STATUS` на сервере. Вы не сможете определить, была ли таблица создана в памяти, а затем скопирована на диск или изначально создана на диске (возможно, из-за столбца `BLOB`), но можете хотя бы увидеть, как часто таблицы копируются на диск. Изучите переменные `Created_tmp_disk_tables` и `Created_tmp_tables`.

- `max_connections`. Эта настройка действует как аварийный тормоз, предохраняя сервер от перегрузки из-за переполнения соединений из приложений. Если приложение работает с ошибкой или сервер сталкивается с проблемой, такой как стопор, может открыться много новых подключений. Но открытое соединение, если оно не может выполнять запросы, абсолютно не нужно, поэтому отказ из-за ошибки «Слишком много соединений» — это способ быстро и просто выйти из строя.

  Установите `max_connections` достаточно высоким для того, чтобы обеспечить обычную рабочую нагрузку, при этом добавьте запас прочности, связанный с дополнительной нагрузкой при входе в систему и администрировании сервера. Например, если вы считаете, что в обычных условиях у вас будет 300 соединений или около того, можете установить значение около 500. Даже если вы не знаете, сколько у вас обычно будет соединений, число 500 ни в коем случае не является

необоснованной отправной точкой. Значение по умолчанию — `100`, но для большинства приложений этого недостаточно.

Остерегайтесь сюрпризов, которые могут заставить вас перейти верхнюю границу числа соединений. Например, если вы перезагрузите сервер приложений, он может не закрыть свои подключения штатно, а MySQL может не понять, что они были закрыты. Когда сервер приложений перезагрузился и пытается заново открыть соединения с базой данных, ему может быть отказано из-за мертвых соединений, время которых еще не истекло.

Постоянно следите за переменной состояния `Max_used_connections`. Это верхняя отметка, которая показывает вам, не испытывает ли сервер в какой-то момент пиковой нагрузки в соединениях. Если значение этой переменной достигает значения переменной `max_connections`, высока вероятность того, что клиенту хотя бы раз было отказано в доступе. Следовательно, стоит применить методики, показанные в главе 3, для фиксации активности сервера.

- `thread_cache_size`. Вы можете вычислить разумное значение для этой переменной, наблюдая за поведением сервера в течение некоторого времени. Следите за переменной состояния `Threads_connected` и определите ее типичные максимум и минимум. Возможно, следует установить кэш потока достаточно большим, чтобы сохранить разницу между пиковыми и непиковыми нагрузками, а может, и чуть больше. Если вы установите его слишком большим, это не станет слишком большой проблемой. Можете задать его размер в два-три раза больше, чем нужно, чтобы справляться с возможными колебаниями. Например, если кажется, что переменная состояния `Threads_connected` варьируется от `150` до `175`, вы можете установить кэш потока в размере `75`. Но, скорее всего, не надо устанавливать его очень большим, поскольку не очень полезно хранить огромное количество запасных потоков, ожидающих подключения; предельное значение `250` — хорошее круглое число (или `256`, если вы предпочитаете степень двойки).

  Вы также можете следить за изменением со временем значения переменной состояния `Threads_created`. Если оно велико или увеличивается — это еще одна подсказка, намекающая на увеличение значения переменной `thread_cache_size`. Проверьте `Threads_cached` и узнайте, сколько потоков уже находится в кэше.

  Связанная переменная состояния — `Slow_launch_threads`. Большое значение этой переменной означает, что что-то задерживает новые потоки при подключении. Это признак того, что с вашим сервером что-то не так, но что именно, непонятно. Обычно это означает: система перегружена, из-за чего операционная система не планирует время процессора для вновь созданных потоков. Это не обязательно означает, что нужно увеличить размер кэша потоков. Проблему следует диагностировать и исправить, а не маскировать ее кэшем, поскольку это может повлиять и на другие процессы.

- `table_cache_size`. Этот кэш (или два кэша, на которые он был разделен в MySQL 5.1) должен быть установлен достаточно большим, чтобы не приходилось открывать и повторно разбирать определения таблиц. Вы можете удостовериться в этом, проверив значение `Open_tables` и изменения в течение времени значения `Opened_tables`. Если вы видите много `Opened_tables` в секунду, возможно, значение

`table_cache` недостаточно велико. Однако созданные явно временные таблицы также могут вызывать рост числа открытых таблиц, даже если кэш таблиц не используется в полной мере, поэтому, возможно, беспокоиться не о чем. Если `Opened_tables` постоянно растет, хотя значение `Open_tables` не так велико, как `table_cache_size`, то это может быть признаком проблемы.

Даже если кэш таблиц полезен, не стоит присваивать этой переменной слишком большое значение. Оказывается, кэш таблиц в двух случаях может приводить к отрицательным результатам.

Во-первых, MySQL использует не слишком эффективный алгоритм для проверки кэша, поэтому, если он действительно велик, это может происходить очень медленно. Вероятно, в большинстве случаев не стоит делать его превышающим 10 000 или 10 240, если вам нравятся степени двойки[1].

Вторая причина избегать установки большого значения состоит в том, что некоторые рабочие нагрузки просто не кэшируемы. Если рабочая нагрузка не кэшируема, то результатом станут непопадания в кэш независимо от того, насколько большим вы его сделаете, забудете про него или установите его в `0`! Это поможет вам избежать ухудшения ситуации: непопадание в кэше лучше, чем затратная проверка кэша, за которой последует непопадание в него. Какие виды рабочих нагрузок не кэшируемы? Если у вас есть десятки или сотни тысяч таблиц и вы используете их в целом равномерно, то, скорее всего, не сможете кэшировать их все и вам лучше установить небольшое значение этой переменной. Иногда это подходит для систем с очень большим количеством размещенных приложений, ни одно из которых не слишком занято.

Разумное начальное значение для этой переменной должно быть в десять раз больше, чем `max_connections`, в большинстве случаев — около 10 000.

Существует несколько прочих параметров, которые часто включаются в конфигурационный файл, в том числе двоичный журнал и настройки репликации. Двоичный журнал полезен для обеспечения восстановления на конкретный момент времени и для репликации, а репликация имеет несколько собственных настроек. Мы расскажем о важных настройках в главах, посвященных репликации и резервному копированию.

Настройки безопасности и готовности к работе

После того как вы установили базовые настройки конфигурации, можете включить несколько параметров, которые делают сервер более безопасным и надежным. Некоторые из них влияют на производительность, поскольку гарантировать безопасность и надежность, как правило, довольно затратно. Однако некоторые из них просто разумны: они защищают от совершения ошибок, таких как, например, вставка

[1] Вы знаете анекдот о степенях двойки? Есть десять типов людей в мире: те, кто понимает двоичную систему счисления, и те, кто — нет. Есть еще десять типов людей: те, кто думает, что двоичные/десятичные шутки смешны, и те, кто занимается сексом. Мы не скажем, считаем ли эти шутки смешными.

бессмысленных данных на сервер. При этом некоторые не влияют на повседневную работу, но защищают от серьезных проблем в экстремальных случаях.

Давайте сначала рассмотрим полезные параметры, характерные для повседневного поведения сервера.

- `expire_logs_days`. Если вы включили двоичный журнал, то должны включить и этот параметр, который задает количество дней, по истечении которых старые двоичные журналы должны очищаться. Если вы не сделаете этого, то в конечном итоге у сервера закончится дисковое пространство и он зависнет или аварийно остановится. Мы рекомендуем установить этот параметр достаточно большим, чтобы можно было восстановить данные, сохраненные как минимум две резервные копии назад (в случае сбоя последней резервной копии). Даже если вы делаете резервные копии каждый день, по-прежнему сохраняйте двоичные журналы как минимум 7–14-дневной давности. Наш опыт показывает, что в случае возникновения необычной проблемы, например при восстановлении подчиненного сервера, а затем при попытке синхронизироваться с главным, вы скажете спасибо за недельный или двухнедельный двоичный журнал. Следует хранить достаточное количество двоичных журналов, чтобы иметь свободу маневра для таких операций.
- `max_allowed_packet`. Этот параметр не позволяет серверу отправлять слишком большой пакет, а также определяет размер принимаемого пакета. Значение по умолчанию, вероятно, слишком мало, но точно так же оно может быть установлено опасно большим. Если оно слишком мало, то могут возникать проблемы при репликации, обычно когда подчиненный сервер не может получить от главного сервера данные, необходимы для репликации. Вы можете увеличить этот параметр до 16 Мбайт или около того.

  Это не задокументировано, но данный параметр также контролирует максимальный размер переменной, определяемой пользователем. Поэтому, если вам нужны очень большие переменные, будьте осторожны: они могут быть усечены или установлены в `NULL`, если превысят размер этой переменной.
- `max_connect_errors`. Если в какой-то момент во время работы с сетью что-то пошло не так, причины проблемы могут быть различными. Это может быть просто ошибка приложения или конфигурации. Другой проблемой может стать недостаток привилегий, препятствующий быстрому успешному завершению соединения. В этом случае клиенты могут попасть в черный список и не смогут подключиться снова, пока вы не очистите кэш хоста. Значение по умолчанию для этого параметра настолько мало, что данная ситуация может очень легко произойти. Вы можете увеличить его, а если уверены, что сервер нельзя взломать простым перебором, то сделать его даже очень большим, тем самым эффективно решив проблему черного списка хостов.
- `skip_name_resolve`. Этот параметр нейтрализует другую ловушку, связанную с сетью и аутентификацией: поиск DNS. DNS является одним из слабейших звеньев в процессе подключения MySQL. Когда вы подключаетесь к серверу, он по умолчанию пытается определить имя хоста, с которого вы подключаетесь, и использует его как часть аутентификации учетных данных. (То есть ваши учетные данные — это имя пользователя, имя хоста и пароль, а не только имя пользователя

и пароль.) Но для проверки имени хоста сервер должен выполнить обратный и прямой DNS-поиск. Все в порядке, пока DNS не столкнется с проблемами, что вполне может произойти. Когда это случается, все накапливается и в конце концов время соединения заканчивается. Чтобы этого избежать, мы настоятельно рекомендуем установить данный параметр, который отключает поиск DNS во время аутентификации. Но если вы этим занимаетесь, необходимо преобразовать разрешения, основанные на имени хоста, на использование IP-адресов джокерных символов или localhost, поскольку аккаунты, основанные на имени хоста, будут отключены.

- `sql_mode`. Этот параметр может принимать различные значения, изменяющие поведение сервера. Мы не рекомендуем менять их только ради любопытства, лучше всего позволить MySQL в большинстве случаев быть самой собой и не пытаться заставить ее вести себя как другие серверы баз данных. (Например, многие клиентские инструменты и инструменты с графическим интерфейсом ожидают, что MySQL будет иметь собственный диалект SQL, поэтому, если вы измените его, сделав более ANSI-совместимым, что-то может пойти не так.) Однако некоторые из этих настроек весьма полезны, а часть из них подходит для специфических случаев. Возможно, стоит посмотреть документацию по следующим параметрам: `STRICT_TRANS_TABLES`, `ERROR_FOR_DIVISION_BY_ZERO`, `NO_AUTO_CREATE_USER`, `NO_AUTO_VALUE_ON_ZERO`, `NO_ENGINE_SUBSTITUTION`, `NO_ZERO_DATE`, `NO_ZERO_IN_DATE` и `ONLY_FULL_GROUP_BY` — и подумать о возможностях их использования.

  Однако имейте в виду, что не стоит менять эти параметры для существующих приложений, поскольку это может привести к несовместимости сервера с ожиданиями приложения. Очень часто люди невольно пишут запросы, в том числе с использованием агрегированных функций, к столбцам, отсутствующим в разделе `GROUP BY`. Поэтому, если вы хотите включить параметр `ONLY_FULL_GROUP_BY`, стоит сделать это сначала на сервере, используемом для разработки, или вспомогательном сервере, убедиться, что все хорошо работает, и только потом разворачивать в «боевых условиях».

- `sysdate_is_now`. Это еще один параметр, который может быть несовместимым с ожиданиями приложений. Но если у вас нет явного желания, чтобы функция `SYSDATE()` имела недетерминированное поведение, которое может нарушить репликацию и сделать ненадежным восстановление из резервных копий, можете включить этот параметр и сделать поведение функции детерминированным.

Несколько параметров контролируют поведение репликации и помогают предотвратить проблему с репликами.

- `read_only`. Этот параметр запрещает непривилегированным пользователям вносить изменения в подчиненные серверы, поскольку они должны получать изменения только от главного сервера, а не от приложения. Мы настоятельно рекомендуем установить для подчиненных серверов режим «только для чтения».
- `skip_slave_start`. Этот параметр не дает MySQL закусить удила и автоматически запустить репликацию. Целесообразно отключать автоматический запуск репликации, поскольку он небезопасен после сбоя или какой-то другой пробле-

мы: человек должен вручную проверить сервер и удостовериться, что начинать репликацию безопасно.

- `slave_net_timeout`. Этот параметр определяет, сколько пройдет времени до того момента, когда подчиненный сервер заметит, что соединение с основным сервером прервано и его требуется установить заново. Значение 1 час, установленное по умолчанию, слишком велико. Установите его равным 1 минуте или даже меньше.
- `sync_master_info`, `sync_relay_log` и `sync_relay_log_info`. Эти параметры, доступные с версии MySQL 5.5, исправляют давние проблемы с подчиненными серверами: они не синхронизируют свои файлы состояния с диском, поэтому, если у главного сервера случится аварийный останов, то останется только догадываться, каково текущее состояние реплики относительно главного сервера. Кроме того, будут повреждены журналы ретрансляции. Эти параметры увеличивают вероятность восстановления реплики после сбоя. По умолчанию они не включены, поскольку вызывают дополнительные операции `fsync()` для реплик, что может их замедлить. Мы предлагаем включить эти параметры, если вы используете добротное оборудование, и отключить их, если есть проблема с репликацией, которую вы можете отследить с задержкой, вызванной функцией `fsync()`.

  В Percona Server есть лучший способ сделать это, включив параметр `innodb_overwrite_relay_log_info`. Тем самым InnoDB сохранит позицию репликации в своем журнале транзакций, который полностью транзакционный и не требует дополнительных операций `fsync()`. Во время восстановления после сбоя InnoDB проверит файлы метаданных репликации и обновит их, если они устарели.

Дополнительные настройки InnoDB

Вспомните рассказ об истории изменений InnoDB в главе 1: сначала она была встроенной подсистемой хранения, затем стала доступной в двух версиях, а теперь новая версия подсистемы снова встроена в сервер. У нового кода InnoDB больше возможностей, и он гораздо сильнее масштабируем. Если вы используете MySQL 5.1, то должны явно настроить ее так, чтобы старая версия InnoDB игнорировалась, а использовалась только более новая. Это значительно улучшит производительность сервера. Следует включить параметр `ignore_builtin_innodb`, а затем включить InnoDB в качестве плагина, настроив соответствующим образом параметр `plugin_load`. Уточните синтаксис вашей платформы в руководстве по InnoDB[1].

Несколько параметров станут доступными в новой версии InnoDB, как только вы ее активизируете. Часть из них очень важна для производительности сервера, другие обеспечивают безопасность и готовность к работе.

- `innodb`. Этот параметр выглядит довольно безобидным, но на самом деле очень важен. Если вы установите значение `FORCE`, сервер не сможет запуститься, если

[1] В Percona Server есть только одна версия InnoDB, и она встроена, поэтому вам не нужно отключать одну версию и загружать взамен нее другую.

не запустится InnoDB. Если вы используете InnoDB как подсистему хранения по умолчанию, это именно то, что вам надо. Действительно, сервер не должен запускаться, если из-за какой-то ошибки, например, в настройках не запускается InnoDB. Дело в том, что приложение, работающее с ошибками, может затем подключиться к серверу, устроить неразбериху и сильно навредить. Будет намного лучше, если сервер просто не запустится, что заставит вас заглянуть в журнал ошибок, а не верить, что с сервером все в порядке.

- `innodb_autoinc_lock_mode`. Этот параметр управляет тем, как InnoDB генерирует автоматическое увеличение значений первичного ключа, что в некоторых случаях может стать узким местом, например в случае вставок с высокой конкурентностью. Если у вас много транзакций, ожидающих блокировки автоинкремента (можете увидеть это в выводе команды `SHOW ENGINE INNODB STATUS`), следует изучить этот параметр. Мы не будем дублировать объяснение этого параметра и его функций, приведенное в руководстве пользователя.
- `innodb_buffer_pool_instances`. В MySQL 5.5 и более новых версиях этот параметр делит буферный пул на несколько сегментов и, вероятно, является одним из наиболее важных способов повышения масштабируемости MySQL на многоядерных машинах с высококонкурентной рабочей нагрузкой. Несколько буферных пулов разделяют рабочую нагрузку, так что некоторые глобальные мьютексы не становятся горячими точками конкуренции.

  Пока неясно, какими принципами следует руководствоваться при выборе количества экземпляров буферного пула. Мы запускали большинство эталонных тестов на серверах с восемью пулами, однако так и не сумели разобраться в некоторых нюансах применения нескольких экземпляров буферного пула. Скорее всего, мы их не поймем, пока MySQL 5.5 не станет повсеместно использоваться в течение более длительного времени.

  Мы не хотим сказать, что MySQL 5.5 редко развертывается в боевых условиях. Дело в том, что самые экстремальные случаи конкуренции мьютексов из тех, что мы помогали решать, встречались у очень больших, очень консервативных пользователей, у которых обновлению предшествовали многие месяцы планирования и проверки. Эти пользователи иногда используют специально настроенную под них версию MySQL, что заставляет их подходить к вопросам обновления в два раза тщательнее. Когда многие из них перейдут на MySQL 5.5 и сбросят на нее уникальную нагрузку, мы, вероятно, узнаем что-то интересное о работе с несколькими буферными пулами. А до тех пор можем лишь сказать, что работа с восемью экземплярами буферного пула кажется очень эффективной.

  Стоит отметить, что Percona Server использует другой подход к решению проблем, связанных с конкуренцией мьютексов InnoDB. Вместо того чтобы разбивать буферный пул — старый проверенный способ, работающий во многих системах, таких как InnoDB, — мы решили разделить ряд глобальных мьютексов на более мелкие специализированные мьютексы. Наши эталонные тесты показывают, что лучший результат достигается при комбинации двух подходов, которая реализована в Percona Server версии 5.5: несколько буферных пулов и более мелкие мьютексы.

- `innodb_io_capacity`. Ранее InnoDB разрабатывалась в предположении, что она работает на одном жестком диске, способном выполнять 100 операций ввода/вывода в секунду. Это была неудачная настройка по умолчанию. Теперь вы можете задать для InnoDB производительность доступной системы ввода/вывода. Иногда необходимо установить довольно большое значение этого параметра (десятки тысяч для чрезвычайно быстрых элементов хранения, таких как flash-устройства PCI-E), для устойчивого сброса «грязных» страниц. Причины этого действия довольно сложно объяснить.
- `innodb_read_io_threads` и `innodb_write_io_threads`. Эти параметры определяют, сколько фоновых потоков доступно для операций ввода/вывода. По умолчанию в последних версиях MySQL есть четыре потока чтения и четыре потока записи, которых вполне достаточно для большинства серверов, особенно при реализации естественного асинхронного ввода/вывода, доступного в MySQL 5.5. Если у вас много жестких дисков, рабочая нагрузка обладает высокой конкурентностью и вы видите, что потоки с трудом справляются с нагрузкой, то можете увеличить количество потоков или даже установить их количество равным числу физических шпинделей, которые используются для ввода/вывода (даже если они расположены за RAID-контроллером).
- `innodb_strict_mode`. Этот параметр заставляет InnoDB в некоторых условиях выдавать ошибки вместо предупреждений, особенно при недопустимых или даже опасных параметрах команды `CREATE TABLE`. Если включите эту параметр, обязательно проверьте команды `CREATE TABLE`: в этом случае может не получиться создать некоторые таблицы, которые ранее отлично создавались. Иногда это огорчает и становится слишком большим ограничением. Вы вряд ли захотите выяснить это лишь тогда, когда будете пытаться восстановить данные из резервной копии.
- `innodb_old_blocks_time`. У InnoDB есть список недавно использованного (least recently used, LRU) двухчастного буферного пула, который предназначен для предотвращения нежелательных запросов от вытесняемых страниц, которые много раз использовались в течение длительного времени. Разовый запрос, например выполненный утилитой mysqldump, как правило, включает страницу в список LRU буферного пула, считывает из нее строки и переходит к следующей странице. Теоретически двухчастный список LRU помешает этой странице вытеснить страницы, которые будут нужны в течение длительного времени. Эти страницы будут помещены в новый подсписок, и только после того, как к ним обратятся несколько раз, они будут перемещены в старый подсписок. Но в InnoDB такое поведение не задано по умолчанию, поскольку страница состоит из нескольких строк и, следовательно, множественный доступ к чтению этих строк со страницы приведет к тому, что она немедленно переместится в старый подсписок, вытесняя страницы, которые требуются в течение длительного времени. Эта переменная определяет количество миллисекунд, которое должно пройти до перехода страницы из новой части списка LRU в старую. Ее значение по умолчанию — `0`, а присвоение ее небольшого значения, такого как `1000` (1 секунда), признано высокоэффективным при эталонном тестировании.

Итоги главы

Изучив эту главу, вы сможете сконфигурировать свой сервер намного лучше, чем предусмотрено настройками по умолчанию. Сервер должен быть быстрым и стабильным, и у вас не должно возникать необходимости подправлять его конфигурацию, кроме как в чрезвычайных обстоятельствах.

Напомним: мы предложили начать с применения нашего образца файла конфигурации, установить основные параметры для ваших сервера и рабочей нагрузки, добавить желаемые параметры безопасности и готовности к работе и при необходимости настроить новые параметры, доступные в плагине InnoDB и MySQL 5.5. Это действительно все, что вам нужно сделать.

Если вы используете InnoDB, как делает большинство, то наиболее важными параметрами являются два:

- `innodb_buffer_pool_size`;
- `innodb_log_file_size`.

Поздравляем — вы только что решили подавляющее большинство проблем конфигурации, с которыми мы встречались на практике! Если вы воспользуетесь нашим инструментом настройки с сайта http://tools.percona.com, то получите хорошую отправную точку для этих и других параметров конфигурации.

Мы также внесли много предложений о том, чего делать не следует. Наиболее важными из них являются следующие:

- не делайте тонкую настройку своего сервера;
- не используйте коэффициенты, формулы или настроечные скрипты в качестве основы для установки переменных конфигурации;
- не доверяйте советам неизвестных людей в Интернете;
- не охотьтесь с командой `SHOW STATUS` наперевес на элементы, которые «плохо выглядят». Если что-то действительно не так, это проявится при профилировании вашего сервера.

Есть несколько важных настроек, которые мы не рассмотрели в этой главе, но которые важны для конкретных типов аппаратных средств и рабочих нагрузок. Мы отложили их обсуждение, потому что считаем, что любые рекомендации по поводу настроек должны быть сопряжены с объяснением того, как протекают внутренние процессы. Это приводит нас к следующей главе, в которой мы покажем, как оптимизировать оборудование и операционную систему для MySQL и наоборот.

9 Оптимизация операционной системы и оборудования

Сервер MySQL в целом работает не лучше, чем самое слабое звено всего аппаратно-программного комплекса, и зачастую лимитирующими факторами являются операционная система и оборудование. Емкость диска, объем доступной памяти, ресурсы процессора, сеть и компоненты, которые связывают все воедино, — все это может ограничивать общую производительность системы. Таким образом, вам необходимо тщательно выбрать оборудование и соответствующим образом настроить его и операционную систему. Например, если при текущей рабочей нагрузке узким местом является ввод/вывод, то можно, конечно, попробовать перепроектировать приложение так, чтобы уменьшить объем подобных операций на уровне MySQL. Но чаще разумнее обновить подсистему ввода/вывода, установить дополнительную память или переконфигурировать существующие диски.

Оборудование меняется очень быстро, поэтому все, что мы напишем о конкретных продуктах или компонентах в этой главе, быстро устареет. Как обычно, наша цель — помочь вам лучше разобраться в протекающих процессах, чтобы вы могли применять свои знания в ситуациях, которые мы напрямую не рассмотрели. Однако будем рассматривать имеющееся в настоящее время оборудование, чтобы проиллюстрировать свои тезисы.

Что ограничивает производительность MySQL

На производительность MySQL могут влиять многие аппаратные компоненты, но чаще всего узким местом оказываются перегрузка процессора и подсистемы ввода/вывода. Перегрузка процессора возникает, когда MySQL работает с данными, которые целиком помещаются в оперативной памяти или могут считываться с диска с необходимой скоростью. При доступных в настоящее время объемах ОЗУ множество наборов данных полностью помещаются в память.

В то же время перегрузка подсистемы ввода/вывода обычно наблюдается, когда нужно прочитать больше данных, чем помещается в память. Если приложение является

распределенным или выполняет очень много запросов и/или требуется небольшая задержка, это узкое место может переместиться в сеть.

Методики, показанные в главе 3, помогут вам найти лимитирующий фактор своей системы, однако не позволяйте себе останавливаться на очевидном, когда покажется, что вы нашли узкое место. Слабое звено в одном сегменте часто воздействует на какую-то другую подсистему, которая и кажется источником проблем. Например, в случае нехватки памяти MySQL может быть вынуждена сбрасывать кэши, чтобы освободить место, а спустя мгновение снова читать только что записанные на диск данные (это относится как к операциям чтения, так и к операциям записи). В таком случае недостаток памяти проявится как насыщение подсистемы ввода/вывода. Когда вы находите компонент, который ограничивает систему, спросите себя: «Является ли этот компонент проблемой, или система предъявляет необоснованные требования к этому компоненту?» Мы рассмотрели этот вопрос в кейсе в главе 3.

Можно привести и другой пример: перегрузка шины памяти может выглядеть как недостаток мощности ЦП. На самом деле, когда мы говорим, что ЦП — узкое место или что приложение ограничено возможностями процессора, мы имеем в виду, что оно занято главным образом вычислениями. В дальнейшем мы подробнее займемся этим вопросом.

Как выбирать процессоры для MySQL

При модернизации или покупке нового оборудования необходимо решить, ограничена ли рабочая нагрузка возможностями процессора.

Определить, какова нагрузка, можно, посмотрев на уровень использования процессора, но лучше не следить за общей загруженностью ЦП, а попытаться изучить соотношение загруженности ЦП и подсистемы ввода/вывода для наиболее важных запросов, обращая особое внимание на то, все ли процессоры загружены равномерно. Чтобы понять, что именно ограничивает производительность сервера, можно воспользоваться инструментами, которые мы рассмотрим чуть позже.

Что лучше: быстрые процессоры или много процессоров

Если рабочая нагрузка приводит к перегрузке ЦП, то, как правило, для MySQL лучше, когда процессоры более быстрые (в противоположность большему количеству процессоров).

Не всегда это так, поскольку все зависит от характера рабочей нагрузки и количества ЦП. Старые версии MySQL плохо масштабировались на большое число процессоров, и даже новые версии MySQL не умеют распараллеливать выполнение одного запроса на несколько ЦП. В результате время обработки запроса с большим объемом вычислений ограничено именно быстродействием процессора.

Когда мы говорим о процессорах, то для упрощения текста несколько вольно обращаемся с терминологией. Современные промышленные серверы обычно имеют несколько сокетов, в каждом из которых есть несколько ядер процессора (которые имеют независимые функциональные блоки), а каждое ядро может иметь несколько аппаратных потоков. Требуется некоторое время, чтобы разобраться в этой сложной архитектуре, и мы не всегда будем придерживаться четкой терминологии. Как правило, говоря о скорости процессора, мы имеем в виду скорость функционального блока, а когда упоминаем количество процессоров, имеем в виду число, которое видит операционная система, хотя оно может быть кратным количеству независимых функциональных блоков.

Современные процессоры значительно улучшены по сравнению с теми, что использовались несколько лет назад. Например, сегодняшние процессоры Intel намного быстрее, чем предыдущие поколения, из-за таких модификаций, как непосредственно подключенная память и улучшенное соединение с такими устройствами, как PCIe-карты. Это особенно актуально для очень быстрых устройств хранения, таких как флеш-накопители Fusion-io и Virident PCIe.

Гиперпоточность также работает намного лучше, чем раньше, а операционные системы сейчас отлично знают, как ее использовать. Раньше считали, что операционные системы не понимают, что два виртуальных процессора действительно находятся на одном и том же кристалле, и планировали задачи для двух виртуальных процессоров, располагающихся на одном и том же физическом функциональном блоке, полагая, что они независимы. Разумеется, один функциональный блок не может одновременно запускать два процесса, поэтому они будут конфликтовать и бороться за ресурсы. Тем не менее операционная система оставила бы остальные CPU бездействующими, понапрасну растрачивая мощность. Операционная система должна использовать гиперпоточность, поскольку должна знать, когда функциональный блок фактически неактивен, и соответствующим образом переключать задачи. Общей причиной возникновения таких проблем было ожидание шины памяти, которая может занимать до 100 циклов процессора и аналогична ожиданию ввода/вывода в очень малом масштабе. В новых операционных системах все это значительно улучшено. Теперь гиперпоточность работает нормально; раньше мы часто советовали людям время от времени отключать ее, но с некоторых пор перестали это делать.

Все это говорит о том, что сейчас у вас может быть много быстрых процессоров — гораздо больше, чем могло быть, когда мы опубликовали второе издание этой книги. Итак, что лучше, много или быстро? Обычно лучше и то и другое. В общем, перед вами могут стоять две цели.

- *Обеспечить для сервера низкую задержку (быстрое время отклика).* Для достижения этой цели требуются быстрые ЦП, так как каждый запрос выполняется только на одном процессоре.
- *Обеспечить для сервера высокую пропускную способность.* Если одновременно выполняется много запросов, то их обслуживание можно ускорить за счет увеличения количества процессоров. Однако на практике это зависит от многих факторов. Так как MySQL с ростом числа процессоров плохо масштабируется, то

существует некий предел числа используемых процессоров. В более старых версиях сервера (приблизительно до поздних выпусков MySQL 5.1) это было серьезным ограничением. В более новых версиях вы можете с уверенностью масштабировать до 16 или 24 процессоров, а может, и дальше в зависимости от того, какую версию используете (здесь у Percona Server есть небольшое преимущество).

Если имеется несколько ЦП, а запросы выполняются не одновременно, то MySQL все же может задействовать дополнительные процессоры для таких фоновых задач, как вытеснение буферов InnoDB, сетевые операции и т. д. Но обычно эти задания — ничто по сравнению с выполнением запросов.

Репликация в MySQL (обсуждается в следующей главе) также выигрывает от наличия быстрых процессоров, а не от того, что их много. Если рабочая нагрузка ограничена мощностью процессора, то распараллеливание ее на главном сервере после сериализации легко может привести к такой нагрузке на подчиненный сервер, с которой тот не справится, даже если он мощнее главного. Впрочем, обычно узким местом на подчиненном сервере оказывается подсистема ввода/вывода, а не процессор.

Если рабочая нагрузка ограничена возможностями процессора, есть и другой подход к вопросу о том, что лучше — более быстрый процессор или большее количество ЦП? Необходимо проанализировать, что именно делают запросы. На аппаратном уровне запрос может либо выполняться, либо находиться в состоянии ожидания. Наиболее распространенными причинами ожидания являются пребывание в очереди на выполнение (когда процесс готов выполняться, но все процессоры заняты), ожидание защелки или блокировки и ожидание завершения дисковой операции или сети. Как вы полагаете, чего ждут запросы? Если они ожидают освобождения защелки или блокировки, то, как правило, помогает увеличение быстродействия процессора; если они стоят в очереди на выполнение, тогда может помочь увеличение как количества процессоров, так и их быстродействия. (Бывают и исключения, например ожидание мьютекса, защищающего буфер журнала InnoDB, который не освобождается до полного завершения операции ввода/вывода, — это может указывать на недостаточную пропускную способность подсистемы ввода/вывода.)

При этом MySQL может эффективно задействовать несколько процессоров для некоторых разновидностей рабочей нагрузки. Пусть, например, имеется множество соединений, выполняющих запросы к разным таблицам (следовательно, не возникает конкуренции за табличные блокировки — проблемы, характерной для таблиц типа MyISAM и Memory), и общая пропускная способность сервера важнее времени отклика для отдельных запросов. При таком сценарии пропускная способность может быть очень высока, поскольку все потоки работают одновременно, не конкурируя между собой.

Но опять-таки заметим, что на практике все может обстоять хуже, чем в теории: в InnoDB имеются глобальные разделенные структуры данных независимо от того, читают ли запросы из отдельных таблиц или нет, а MyISAM имеет глобальные блокировки в каждом буфере ключей. Это не только подсистемы хранения — во всем винили InnoDB, но после некоторых улучшений проявились другие узкие места на более высоких уровнях на сервере. Печально известный мьютекс LOCK_open может

быть реальной проблемой для MySQL 5.1 и более старых версий, то же самое можно сказать о некоторых других мьютексах на уровне сервера (например, о кэше запросов).

Обычно эти типы конкуренции можно диагностировать с помощью трассировки стека (см., например, утилиту pt-pmp в пакете Percona Toolkit). Если вы столкнулись с такими проблемами, возможно, стоит изменить конфигурацию сервера, отключив или изменив компонент-нарушитель, секционировать (шардировать) свои данные или что-то изменить в своей работе. В реальности может возникнуть так много проблем, которые будут требовать от нас соответствующих решений, что перечислить их все не представляется возможным. Однако, к счастью, ответ обычно становится очевиден после того, как установлен верный диагноз. К счастью, большинство проблем — это крайние случаи, с которыми вы вряд ли столкнетесь, а наиболее распространенные случаи исправляются с течением времени в самом сервере.

Архитектура ЦП

Вероятно, более 99 % экземпляров сервера MySQL (исключая встроенное использование) запускаются на архитектуре x86 на чипах Intel или AMD. В этой книге мы по большей части будем рассматривать именно такую архитектуру.

Сейчас повсеместно используются 64-разрядные архитектуры, и купить 32-разрядный процессор очень трудно. MySQL неплохо работает на 64-разрядных процессорах, хотя некоторые его внутренние компоненты еще не переписаны для полной поддержки такой архитектуры, поэтому, если вы используете старую версию сервера, возможно, вам надо быть осторожными. Например, в ранних версиях MySQL 5.0 объем буфера ключей MyISAM ограничен 4 Гбайт — это размер, который можно адресовать 32-разрядным числом (впрочем, для преодоления этого ограничения можно создать несколько буферов ключей).

Убедитесь, что вы используете 64-разрядную операционную систему на 64-разрядном оборудовании. В наши дни это встречается реже, чем прежде, но ранее большинство хостинг-провайдеров установили 32-разрядные операционные системы на серверы, даже если на серверах были 64-разрядные процессоры. Это означает, что они не могли использовать большой объем памяти: хотя некоторые 32-разрядные системы и поддерживают очень большой объем памяти, они все же неспособны работать с ней столь же эффективно, как 64-разрядная система, и не один процесс не сможет адресоваться более чем к 4 Гбайт памяти.

Масштабирование на несколько процессоров и ядер

Где несколько ЦП можно задействовать с пользой, так это в системах оперативной обработки транзакций (OLTP). В них обычно приходится иметь дело с большим количеством мелких операций, которые можно выполнять на разных процессорах, поскольку они поступают по разным соединениям. В таких условиях конкурентный доступ становится узким местом. К этой категории систем относится большинство веб-приложений.

На серверах для оперативной обработки транзакций обычно применяется подсистема хранения InnoDB, в которой есть ряд нерешенных проблем с конкурентным доступом при наличии нескольких ЦП. Однако узким местом может стать не только InnoDB. Любой разделяемый ресурс — потенциальный источник конкуренции. InnoDB привлекает столь пристальное внимание просто потому, что чаще всего используется в условиях высокого конкурентного доступа, но MyISAM выглядит ничуть не лучше, если ее как следует нагрузить, даже без какого-либо изменения данных. Многие узкие места, относящиеся к конкурентному доступу, например блокировки строк в InnoDB и табличные блокировки в MyISAM, невозможно устранить в принципе, единственное решение — выполнить операцию как можно быстрее, чтобы освободить блокировку и позволить работать тем потокам, которые ее ожидают. Неважно, сколько имеется процессоров, если все они вынуждены ждать освобождения одной-единственной блокировки. Таким образом, даже при наличии рабочей нагрузки с высокой степенью конкурентности иногда удается получить выигрыш от наличия более быстрых процессоров.

На самом деле в базах данных есть два вида проблем, относящихся к конкурентному доступу, и для их решения нужно применять разные подходы.

- *Логическая конкуренция*. Конкуренция за ресурсы, видимые приложению, например за блокировки на уровне строки или таблицы. Для решения подобных проблем требуются такие мероприятия, как, например, изменение логики приложения, использование другой подсистемы хранения, изменение конфигурации сервера или применение других блокировочных подсказок оптимизатору или уровней изоляции транзакций.
- *Внутренняя конкуренция*. Конкуренция за такие ресурсы, как семафоры, доступ к страницам буферного пула InnoDB и т. д. Можно попробовать найти обходное решение путем изменения конфигурации сервера, параметров операционной системы или установки другого оборудования, но, как правило, вам придется смириться с имеющимися ограничениями. Иногда проблему удается смягчить, воспользовавшись другой подсистемой хранения или наложив заплату на уже используемую.

Количество процессоров, которые MySQL может эффективно задействовать, и масштабируемость MySQL при возрастании нагрузки — паттерн масштабируемости — зависят как от характера рабочей нагрузки, так и от системной архитектуры. Под системной архитектурой мы понимаем операционную систему и оборудование, а не приложение, в котором используется MySQL. На паттерн масштабируемости MySQL оказывают влияние такие факторы, как архитектура процессора (RISC, CISC, глубина конвейера и т. д.), модель процессора и операционная система. Именно поэтому так важно эталонное тестирование: некоторые системы продолжают прекрасно работать при увеличении степени конкурентности, тогда как показатели других резко ухудшаются.

Иногда увеличение количества процессоров приводит даже к ухудшению общей производительности. Причем это весьма распространенная проблема: мы знаем многих, кто пытался апгрейдить четырехъядерную систему до восьмиядерной, но был вынуж-

ден вернуться к старой системе (или предоставить в распоряжение процесса MySQL только четыре ядра из восьми) из-за снижения производительности. В MySQL 5.0 до появления патчей Google, а затем и Percona Server магическим числом были четыре ядра, но сейчас мы видим людей, работающих на серверах с 80 «процессорами» (по данным операционной системы). Планируя большое обновление, проверьте свое оборудование, версию сервера и рабочую нагрузку.

Некоторые узкие места, препятствующие масштабируемости, находятся в самом сервере, тогда как другие — на уровне подсистем хранения. Очень важно, для чего проектировалась конкретная подсистема хранения: иногда переход на другую подсистему позволяет более эффективно задействовать большее количество ЦП.

Войны за увеличение быстродействия ЦП, полыхавшие на рубеже столетий, сейчас несколько поутихли, производители процессоров теперь больше озабочены увеличением количества ядер и такими вариациями на эту тему, как гиперпоточность. Вполне может оказаться, что процессор будущего станет обладать несколькими сотнями ядер, уже сегодня четырех- и шестиядерный ЦП не вызывают удивления. Внутренняя архитектура процессоров разных производителей настолько сильно различается, что невозможно высказать общие соображения о взаимодействии между потоками, процессорами и ядрами. Очень важно и то, как спроектированы память и шина доступа к ней. Ну и наконец, от архитектуры зависит и ответ на вопрос, что лучше — несколько ядер или несколько физических процессоров.

Стоит упомянуть еще две особенности современных процессоров. Первая — частотное масштабирование. Это мощная методика управления, которая динамически изменяет тактовую частоту процессора в зависимости от спроса на ЦП. Проблема в том, что он иногда не очень хорошо справляется с трафиком, состоящим из пакетов коротких запросов, поскольку операционной системе может потребоваться некоторое время, чтобы решить, что процессоры должны тикать с запаздыванием. В результате запросы могут временно работать на более низкой скорости с увеличенным временем отклика. Частотное масштабирование может замедлить работу при неравномерных нагрузках, а также, что более важно, дестабилизировать производительность.

Вторая особенность — технология турбоускорения, которая меняет наше представление о процессорах. Мы привыкли думать, что у нашего четырехъядерного процессора с тактовой частотой 2 ГГц есть четыре одинаково мощных ядра независимо от того, простаивают они или нет. И поэтому вполне масштабируемая система может добиться в четыре раза большей производительности при использовании всех четырех ядер. Но на самом деле это не так: когда система использует только одно ядро, процессор может работать с более высокой тактовой частотой, например 3 ГГц. Это может помешать реализации планов по увеличению мощности и масштабируемости, поскольку система ведет себя нелинейно. Это также означает, что простаивающий процессор не является потраченным впустую ресурсом. Если у вас есть сервер, который в одном потоке запускает репликацию, а также имеются три простаивающих процессора, которые, как вы считаете, можно использовать для решения других задач и это не повлияет на репликацию, то вы, скорее всего, ошибаетесь.

Поиск баланса между памятью и дисками

Иметь много памяти важно прежде всего не для того, чтобы уместить в ней как можно больше данных, а для того, чтобы избежать дисковых операций ввода/вывода, которые на несколько порядков медленнее, чем доступ к оперативной памяти. Хитрость в том, чтобы найти правильный баланс между объемом оперативной и дисковой памяти, быстродействием, затратами и другими характеристиками, которые обеспечили бы высокую производительность при данной рабочей нагрузке. Но прежде, чем заняться решением этой задачи, ненадолго вернемся к основам.

Любой компьютер содержит пирамиду кэшей, причем на каждом уровне объем уменьшается, а быстродействие и затраты растут (рис. 9.1).

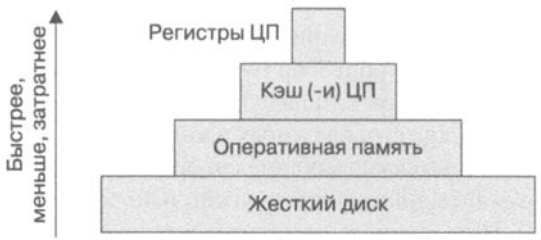


Рис. 9.1. Иерархия кэшей

Чем выше в иерархии находится кэш, тем он лучше приспособлен для хранения часто используемых данных и увеличения скорости доступа к ним. Обычно применяются такие эвристические правила, как «данные, к которым обращались недавно, скорее всего, потребуются снова» и «данные, расположенные близко к недавно использованным, вероятно, тоже скоро понадобятся». Подобные правила работают благодаря пространственной и временной *локальности ссылок*.

Для программиста регистры ЦП и кэши прозрачны и архитектурно зависимы. Ими управляют компилятор и сам процессор. Однако разница между оперативной памятью и жестким диском для программиста весьма ощутима, и программы работают с этими видами памяти совершенно по-разному[1].

В особенности это относится к серверам баз данных, поведение которых зачастую идет вразрез с предсказаниями, выполненными по вышеупомянутым эвристическим правилам. Удачно спроектированный кэш базы данных (такой как буферный пул в InnoDB) обычно оказывается эффективнее кэша операционной системы, ориентированного на задачи общего характера. Кэш базы данных знает о самих данных гораздо больше, и его логика ориентирована на обслуживание специфических потреб-

[1] Впрочем, программы могут полагаться на то, что операционная система кэширует в памяти большой объем данных, которые концептуально находятся на диске. Именно так, кстати, и поступает MySQL. Она рассматривает файлы данных как постоянно находящиеся на дисках и позволяет операционной системе заботиться об их кэшировании для ускорения доступа.

ностей базы данных (например, порядка записи). Кроме того, для доступа к данным, хранящимся в кэше базы данных, не нужен системный вызов.

Из-за названных особенностей специализированного кэша вы должны настраивать иерархию кэшей в соответствии с типичными паттернами доступа к данным в базе. Поскольку регистры и кэши на материнской плате не могут быть сконфигурированы пользователем, то в вашем распоряжении остаются только оперативная память и жесткий диск.

Произвольный и последовательный ввод/вывод

В серверах баз данных применяется как последовательный, так и произвольный ввод/вывод, причем наибольший выигрыш от кэширования получается в последнем случае. В этом легко убедиться, мысленно представив себе смешанную рабочую нагрузку, в которой сочетаются операции поиска одиночных строк и поиска по диапазону, возвращающему множество строк. Как правило, часто используемые данные распределены случайным образом, поэтому их кэширование позволяет избежать затратных операций поиска на диске. Напротив, при последовательном доступе данные обычно считываются только один раз, поэтому кэшировать их бессмысленно, кроме случая, когда они целиком помещаются в память.

Операции последовательного чтения мало выигрывают от кэширования еще и потому, что они выполняются быстрее операций произвольного чтения. Тому есть две причины.

- *Последовательный ввод/вывод быстрее произвольного.* Последовательные операции выполняются быстрее произвольных и в оперативной памяти, и на диске. Предположим, что диск способен выполнить 100 произвольных операций ввода/вывода в секунду и последовательно прочитать 50 Мбайт в секунду (примерно такие показатели характерны для большинства современных дисков потребительского класса). Если длина строки составляет 100 байт, то при произвольном доступе можно будет прочитать 100 строк в секунду, а при последовательном — 500 000. Разница в 5000 раз, то есть на несколько порядков. Поэтому при таком сценарии кэширование результатов операций произвольного доступа оказывается гораздо полезнее.

  Последовательный доступ к строкам в оперативной памяти также быстрее произвольного. Современные микросхемы памяти обычно способны прочитать в секунду примерно 250 000 строк длиной 100 байт при доступе с произвольной выборкой или 5 миллионов таких же строк последовательно. Обратите внимание на то, что произвольный доступ к памяти в 2500 раз быстрее такого же доступа к диску, тогда как последовательный быстрее всего лишь в десять раз.

- *Подсистемы хранения могут выполнять последовательное чтение быстрее произвольного.* Для произвольного доступа подсистема хранения обычно должна воспользоваться индексом. (Из этого правила есть исключения, но для InnoDb и MyISAM оно справедливо.) Чаще всего это требует навигации по B-дереву

и сравнения значений. В то же время последовательное чтение обычно сводится к обходу гораздо более простой структуры данных, например связанного списка. Работы в этом случае гораздо меньше, а значит, последовательное чтение происходит быстрее.

Наконец, произвольное чтение обычно выполняется для поиска отдельных строк. Но чтение — это не только одна строка, это целая страница данных, большинство из которых не нужны. Что означает большой объем работы, проделанной впустую. В то же время последовательное чтение, как правило, используется для извлечения всех строк страницы, поэтому с точки зрения затрат оно было более эффективно.

Кэширование результатов последовательного чтения позволяет кое-что сэкономить, но эффективность кэширования результатов произвольного чтения многократно выше. Иными словами, для решения проблем, возникающих из-за ввода/вывода с произвольным доступом, лучше всего добавить память, если такая возможность существует.

Кэширование, чтение и запись

Если памяти достаточно, то при чтении можно вообще обойтись без доступа к диску. Коль скоро все данные помещаются в памяти, то после «прогрева» сервера любая операция чтения будет удовлетворяться из кэша. Логические операции чтения, конечно же, останутся, но физических не будет. Однако с записью все обстоит по-другому. Операцию записи, как и операцию чтения, можно выполнить в памяти, но рано или поздно данные должны быть зафиксированы на диске. Другими словами, кэш позволяет отложить запись на диск, но не устранить ее полностью, как в случае чтения.

Помимо откладывания записи, кэш позволяет группировать несколько операций двумя способами.

- *Много операций записи, одна операция сброса.* Один и тот же элемент данных может быть многократно изменен в памяти без записи на диск всех новых значений. Когда в итоге данные все же сбрасываются на диск, то фиксируется результат всех модификаций, произошедших с момента последней физической записи. Например, хранящийся в памяти счетчик может обновляться несколькими командами. Если он был увеличен в 100 раз, а затем записан на диск, то получится, что 100 изменений сгруппированы в одну операцию физической записи.
- *Объединение операций ввода/вывода.* Можно модифицировать в памяти несколько разных элементов данных, а затем собрать все изменения вместе и выполнить их одной операцией физической записи на диск.

Именно поэтому во многих транзакционных системах применяется *упреждающая запись в журнал*. Эта технология позволяет выполнять изменения на страницах, хранящихся в оперативной памяти, не сбрасывая их на диск, так как последнее потребовало бы операций ввода/вывода с произвольным доступом, что протекает очень медленно. Вместо этого протокол изменений записывается в последовательный файл

журнала — данная операция выполняется гораздо быстрее. Впоследствии фоновый поток записывает модифицированные страницы в нужное место, причем попутно может оптимизировать запись.

Существенно ускорить операции записи позволяет буферизация, поскольку она преобразует произвольный ввод/вывод в последовательный. Асинхронная (буферизованная) запись обычно возлагается на операционную систему, которая может производить пакетный сброс на диск наиболее оптимальным образом. Синхронные (небуферизованные) записи должны быть занесены на диск до их завершения. Именно поэтому такой выигрыш дает буферизация на уровне RAID-контроллера, оборудованного кэшем записи с резервным электропитанием (мы обсудим технологию RAID в дальнейшем).

Что такое рабочее множество

У каждого приложения есть рабочее множество данных, то есть те данные, которые ему реально нужны для работы. В большинстве баз данных имеется также информация — и много, — не входящая в рабочее множество. Базу можно представлять себе как стол с выдвижными ящиками. Рабочее множество состоит из тех документов, которые должны лежать на столе для работы. В этом случае поверхность стола является аналогом оперативной памяти, а ящики — аналогами жестких дисков. Чтобы выполнить свою работу, вам вовсе не обязательно выкладывать на стол все бумажки. Точно так же для достижения оптимальной производительности не нужно загружать в память всю базу данных — достаточно рабочего множества.

Размер рабочего множества серьезно зависит от приложения. Для одних программ рабочее множество составляет всего 1 % от общего объема данных, а для других приближается к 100 %. Если рабочее множество не умещается целиком в памяти, то сервер вынужден перемещать информацию между диском и памятью. Поэтому нехватка памяти может выглядеть как проблема с вводом/выводом. Иногда поместить все рабочее множество в память в принципе невозможно, а иногда это и не нужно (например, если приложение занято преимущественно последовательным вводом/выводом). В целом архитектура приложения может существенно зависеть от того, умещается ли рабочее множество целиком в оперативную память.

Рабочее множество можно определить как основанный на времени процентиль. Например, 95-й процентиль одночасового рабочего множества — это набор страниц, которые база данных использует в течение 1 часа, за исключением 5 % страниц, которые задействуются реже прочих. Процентиль — это наиболее полезный способ оценки рабочего множества, поскольку возможно, что в течение каждого часа выполняется доступ только к 1 % всех данных, но за 24 часа это может вылиться в 20 %. Быть может, полезнее рассуждать о рабочем множестве как об объеме данных, которые необходимо кэшировать для того, чтобы рабочая нагрузка была ограничена лишь процессорными мощностями. Если вы не можете поместить в кэш достаточно информации, значит, рабочее множество не умещается в памяти.

О рабочем множестве следует говорить в терминах наиболее часто используемого набора страниц, а не наиболее часто читаемого или записываемого. Это означает, что определение рабочего множества требует нужных инструментов внутри приложения: вы не можете просто смотреть на внешнее использование, такое как доступ к вводу/выводу, поскольку ввод/вывод на страницы — это не то же самое, что логический доступ к страницам. MySQL может прочитать страницу в память, а затем получить доступ к ней миллионы раз, но вы увидите только одну операцию ввода/вывода, если, например, будете использовать утилиту strace. Отсутствие инструментов, необходимых для определения рабочего множества, вероятно, является самой значимой причиной того, что на эту тему выполнено не так уж много исследований.

Рабочее множество состоит из данных и индексов, а исчислять его следует в *единицах кэширования*. Единицей кэширования называется наименьший объем данных, которым может оперировать подсистема хранения.

У разных подсистем хранения единицы кэширования различны, поэтому различен и размер рабочего множества. Например, InnoDB по умолчанию оперирует страницами размером 16 Кбайт. Если при поиске одиночной строки InnoDB должна обратиться к диску, то в буферный пул будет считана и закэширована вся содержащая ее страница. Иногда это расточительно.

Предположим, что произвольным образом считываются 100-байтные строки. InnoDB придется использовать много дополнительной памяти в буферном пуле для этих строк, поскольку ей придется читать и кэшировать полную страницу размером 16 Кбайт для каждой строки. А так как в рабочее множество входят и индексы, то InnoDB вынуждена будет прочитать и закэшировать также те части дерева индекса, которые нужны для поиска строки. Длина индексных страниц в InnoDB также составляет 16 Кбайт, следовательно, для доступа всего к одной 100-байтной строке может потребоваться сохранить в кэше 32 Кбайт, а то и больше — в зависимости от глубины дерева индекса. Поэтому единица кэширования — это еще одна причина, по которой в InnoDB так важно правильно выбирать кластерные индексы. Кластерный индекс позволяет не только оптимизировать доступ к диску, но и хранить взаимосвязанные данные в пределах одной страницы, из-за чего в кэше помещается более весомая доля рабочего множества.

Определение эффективного соотношения «память — диск»

Приемлемое соотношение объема оперативной и дисковой памяти лучше всего определять путем экспериментирования и/или эталонного тестирования. Если можно загрузить в оперативную память вообще все, то думать больше не о чем. Но, как правило, это не так, поэтому необходимо выполнить эталонные тесты для некоторого подмножества данных и посмотреть, что получится. Ваша цель — найти приемлемый коэффициент непопадания в кэш. Непопадание наблюдается, когда для выполнения запроса нужны данные, отсутствующие в кэше, так что серверу приходится читать их с диска.

Коэффициент непопадания в кэш определяет то, насколько эффективно задействован процессор, поэтому для его оценки проще всего взглянуть на показатели использования ЦП. Например, если 99 % всего времени ЦП работает, а оставшийся 1 % ожидает завершения ввода/вывода, то коэффициент непопадания можно считать хорошим.

Рассмотрим, как рабочее множество влияет на коэффициент непопадания. Важно понимать, что рабочее множество не просто число, в действительности оно представляет собой статистическое распределение, по отношению к которому коэффициент непопадания нелинеен. Например, если имеется 10 Гбайт памяти, а коэффициент непопадания составляет 10 %, то может показаться, что стоит добавить еще 11 % памяти[1] — и коэффициент непопадания обратится в 0. Но на самом деле из-за таких деталей, как размер единицы кэширования, для достижения коэффициента непопадания 1 % может потребоваться целых 50 Гбайт памяти. И даже при точном совпадении с единицей кэширования теоретическая оценка может оказаться неверной: ситуация нередко осложняется, например, из-за порядка доступа к данным. Чтобы получить коэффициент непопадания 1 %, в зависимости от рабочей нагрузки может потребоваться и 500 Гбайт памяти!

Очень легко увлечься оптимизацией того, что не дает большого выигрыша. Например, значение коэффициента непопадания 10 % может означать, что ЦП занят 80 % времени, а это совсем неплохо. Предположим, что путем добавления памяти вы смогли сократить коэффициент непопадания до 5 %. Сильно упрощая картину, можно сказать, что вы подбросили процессору еще 6 % данных. Пойдя еще на одно упрощение, скажем, что вы довели коэффициент использования процессора до 84,8 %. Но это не такая уж большая победа, если принять во внимание деньги, потраченные на приобретение необходимой для этого памяти. К тому же на самом деле различия в скорости доступа к памяти и диску, характер обработки данных процессором и целый ряд других факторов могут привести к тому, что снижение коэффициента непопадания до 5 % не изменит коэффициента использования ЦП.

Поэтому мы в самом начале сказали, что стремиться нужно к приемлемому коэффициенту непопадания в кэш, а не к нулевому. Невозможно точно указать желаемое значение, поскольку, что считать приемлемым, зависит от конкретного приложения и рабочей нагрузки. Некоторые задачи прекрасно ведут себя при коэффициенте непопадания 1 %, тогда как для нормальной работы других необходим коэффициент 0,01 %. («Хороший коэффициент непопадания в кэш» — расплывчатое понятие, а тот факт, что подсчитать его можно разными способами, только усложняет дело.)

Оптимальное соотношение «память — диск» зависит и от других компонентов системы. Предположим, что в вашем компьютере 16 Гбайт оперативной памяти, 20 Гбайт данных и очень много свободного места на диске. Система прекрасно работает, и процессор загружен на 80 %. Если вы захотите увеличить объем данных вдвое, сохранив уровень производительности, возможно, для этого вы решите удвоить количество

[1] Именно 11, а не 10 %. Если коэффициент непопадания равен 10 %, то коэффициент попадания — 90 %, поэтому необходимо разделить 10 Гбайт на 90 %, что дает 11,111 Гбайт.

процессоров и объем памяти. Однако даже если все компоненты системы идеально масштабируются с учетом возросшей нагрузки (совершенно нереалистичное допущение), то такое решение, скорее всего, ничего не даст. Система, в которой хранится 20 Гбайт данных, вероятно, задействует более 50 % пропускной способности какого-то компонента, например, не исключено, что количество операций ввода/вывода уже составляет 80 % от максимума. Очередь внутри системы также нелинейная. Справиться с удвоенной нагрузкой сервер уже не сможет. Таким образом, наилучшее соотношение «память — диск» определяется самым слабым компонентом системы.

Выбор жестких дисков

Если нужное количество данных поместить в память не удается, например, анализ показал, что при имеющейся подсистеме ввода/вывода для полной загрузки процессора необходимо 500 Гбайт памяти, то стоит подумать о приобретении более мощной подсистемы ввода/вывода, пусть даже за счет памяти. При этом приложение нужно проектировать с учетом задержек ввода/вывода.

Этот подход может показаться противоречащим интуиции. Ведь совсем недавно мы говорили, что дополнительная память может снизить нагрузку на подсистему ввода/вывода и сократить время ожидания. Так зачем же увеличивать мощность этой подсистемы, если проблему можно решить добавлением памяти? Ответ заключается в том, что необходимо соблюдать баланс между различными факторами: соотношением операций чтения и записи, длиной каждой операции ввода/вывода, количеством таких операций в секунду. Например, если нужно, чтобы запись в журнал выполнялась быстро, то невозможно исключить из уравнения диск простым увеличением объема оперативной памяти. В таком случае лучше потратить деньги на приобретение высокопроизводительной подсистемы ввода/вывода, оборудованной кэшем записи с резервным питанием.

Напомним, что операция считывания с обычного жесткого диска состоит из трех шагов.

1. Подвести головку считывания к нужной дорожке.
2. Ждать, пока в результате вращения диска нужные данные не окажутся под головкой.
3. Ждать, пока все нужные данные не пройдут под головкой.

Скорость выполнения этих операций диском оценивается двумя показателями: *временем доступа* (совместно шаги 1 и 2) и *скоростью передачи*. Эти же два числа определяют *задержку* и *пропускную способность*. Что важнее, время доступа, скорость передачи или сочетание того и другого, зависит от характера выполняемых запросов. Если говорить о полном времени, необходимом для завершения чтения с диска, то время поиска произвольной одиночной строки определяется в первую очередь шагами 1 и 2, а последовательное считывание большого объема данных — шагом.

Есть еще ряд факторов, которые могут повлиять на выбор дисков, а какие из них существенны, зависит от конкретного приложения. Давайте представим, что вы выбираете диски для какого-нибудь онлайнового приложения, например популярного новостного сайта, для которого характерно большое количество операций произвольного чтения, возвращающих сравнительно мало данных. Тогда стоит принять во внимание следующие факторы.

- *Емкость диска.* Для онлайновых приложений это редко бывает камнем преткновения, так как емкости современных дисков более чем достаточно. Если это не так, то общепринятой практикой является объединение дисков в RAID-массив[1].
- *Скорость передачи.* Мы уже видели, что современные диски способны передавать данные очень быстро. Точное значение зависит главным образом от скорости вращения шпинделя и плотности записи данных на поверхности диска, а также от ограничений интерфейса с хост-компьютером (многие современные диски могут читать данные быстрее, чем интерфейс способен их передавать). Так или иначе, не скорость передачи лимитирует производительность онлайновых приложений, поскольку последние обычно выполняют много коротких операций с произвольным доступом.
- *Время доступа.* Обычно именно этот параметр определяет производительность произвольной выборки, поэтому нужно искать диски с минимальным временем доступа.
- *Скорость вращения шпинделя.* Сейчас наиболее часто встречаются диски с частотой вращения 7200, 10 000 и 15 000 оборотов в минуту. Скорость как последовательной, так и произвольной выборки в немалой степени зависит от скорости вращения.
- *Габариты.* При прочих равных условиях габариты диска тоже играют роль: чем он меньше, тем меньше времени занимает перемещение головки для считывания. Диски диаметром 2,5 дюйма, предназначенные для серверов, часто оказываются быстрее своих более крупных собратьев. К тому же они потребляют меньше электроэнергии, и в один системный блок помещается больше дисков.

Как и в случае с ЦП, возможность масштабирования MySQL на несколько дисков зависит от подсистемы хранения и от рабочей нагрузки. InnoDB обычно хорошо масштабируется на множество жестких дисков. Однако степень масштабирования подсистемы MyISAM при записи ограничивается табличными блокировками, поэтому если для рабочей нагрузки, включающей много операций записи, используется MyISAM, то существенного выигрыша от наличия большого количества дисков не добиться. В какой-то мере могут помочь буферизация на уровне операционной системы и фоновое распараллеливание записи, но все же масштабируемость записи в MyISAM принципиально хуже, чем в InnoDB.

---

[1] Интересно отметить, что некоторые сознательно покупают большие диски, но используют лишь 20–30 % их емкости. Тем самым повышается локальность данных и сокращается время подвода головки, что иногда оправдывает более высокую цену.

Как и в случае с процессорами, больше дисков — не всегда лучше. Некоторые приложения, для которых важна низкая задержка, нуждаются в более быстрых дисках, а не в большем их количестве. Например, производительность репликации обычно выше при использовании скоростных дисков, поскольку обновления на подчиненном сервере производятся одним потоком.

Твердотельные хранилища данных

На самом деле технологии твердотельных (флеш-память) хранилищ данных уже 30 лет, но она стала хитом сезона, поскольку за последние несколько лет появилось новое поколение устройств. Твердотельные хранилища теперь стали довольно дешевыми и гораздо более усовершенствованными, так что они широко применяются уже сейчас, а в ближайшем будущем, скорее всего, заменят традиционные жесткие диски.

Твердотельные хранилища данных используют энергонезависимые чипы флеш-памяти, состоящие из ячеек, вместо магнитных блинов жесткого диска. Они также называются NVRAM, или *энергонезависимой оперативной памятью*. У них нет движущихся частей, что заставляет их вести себя не так, как жесткие диски. Мы рассмотрим различия в деталях.

Современные технологии, представляющие интерес для пользователей MySQL, можно разделить на две основные категории: SSD (solid-state drives — твердотельные диски) и PCIe-карты. SSDs эмулируют стандартные жесткие диски, реализуя SATA-интерфейс (Serial Advanced Technology Attachment — последовательный интерфейс обмена данными с накопителями информации), поэтому они являются упрощенной заменой жесткого диска, который сейчас находится на вашем сервере, и могут вставляться в существующие слоты в корпусе. В картах PCIe используются специальные драйверы операционной системы, которые представляют хранилище в виде блок-устройства. PCIe- и SSD-устройства иногда называют просто SSD.

Приведем краткое резюме производительности флеш-устройств. У высококачественного флеш-устройства:

- гораздо выше производительность произвольного чтения и записи по сравнению с жесткими дисками. Флеш-устройства обычно немного лучше читают, чем записывают;
- лучшая производительность последовательного чтения и записи, чем у жестких дисков. Тем не менее это не столь впечатляющее улучшение, как в случае произвольного ввода/вывода, поскольку жесткие диски выполняют произвольный ввод/вывод намного медленнее, чем последовательный. На самом деле SSD с минимальной конфигурацией могут быть даже медленнее, чем обычные диски;
- гораздо лучшая поддержка конкурентного доступа, чем у жестких дисков. Флеш-устройства могут поддерживать намного больше параллельных операций, и они не достигнут максимальной пропускной способности, пока у вас не будет большого количества таких операций.

Наиболее важным является улучшение произвольного ввода/вывода и конкурентного доступа. Флеш-память обеспечивает очень хорошую производительность произвольного ввода/вывода при высоком уровне конкурентности, что точно соответствует потребностям нормализованных баз данных. Одной из наиболее распространенных причин денормализации схемы является предотвращение произвольного ввода/вывода и возможность использования последовательного ввода/вывода для обслуживания запросов.

В результате мы полагаем, что твердотельные хранилища данных в будущем фундаментально изменят технологию реляционных СУБД. Современное поколение технологии реляционных СУБД претерпело множество оптимизаций на протяжении десятилетий, чтобы повысилась эффективность хранения в системах на основе шпинделя. Для твердотельных хранилищ пока не существует таких же глубоких исследований и инженерных разработок[1].

Обзор флеш-памяти

У жестких дисков с вращающимися блинами и пишущими головками есть свои ограничения и характеристики, являющиеся следствием законов физики. То же самое можно сказать и о твердотельном хранилище данных, которое построено поверх флеш-памяти. Не думайте, что твердотельное хранилище данных — простое устройство. На самом деле оно сложнее, чем жесткий диск. Ограничения флеш-памяти довольно серьезны, и их трудно преодолеть, поэтому типичное твердотельное устройство имеет сложную архитектуру с множеством абстракций, кэшированием и собственной магией.

Самой важной характеристикой флеш-памяти является то, что ее можно читать много раз на большой скорости и мелкими порциями, но записывать на нее намного сложнее. Вы не можете переписать ячейку[2] без специальной операции стирания, при этом стирать можно только большие блоки, например 512 Кбайт. Цикл стирания медленный и в конечном итоге изнашивает блок. Количество циклов стирания, которое может выдержать блок, зависит от используемой технологии (поговорим об этом позже).

Ограничение на запись является причиной сложности твердотельного хранилища. Вот поэтому одни устройства обеспечивают стабильную производительность, а другие — нет. Все зависит от соответствующей прошивки, драйверов и других элементов, которые запускают твердотельное устройство. Для обеспечения хорошей записи и защиты от преждевременного износа блоков флеш-памяти устройство должно иметь возможность перемещать страницы и выполнять сбор мусора и так называемый

[1] Некоторые компании утверждают, что они с чистого листа начинают разработки, свободные от оков шпиндельного прошлого. Это вызывает определенный скепсис: решать задачи реляционных СУБД непросто.

[2] Это упрощение, но детали здесь и не важны. Если хотите, можете подробнее прочитать об этом в «Википедии».

контроль равномерности износа. Термин *«усиление записи»* используется для описания дополнительных записей, вызванных перемещением данных с места на место, многократной записи данных и образования метаданных из-за частичной записи блоков. Если интересно, посмотрите статью «Википедии» об усилении записи — из нее вы можете узнать больше.

Очень важно понять, в чем заключается процесс сбора мусора. Для того чтобы поддерживать блоки чистыми и готовыми к записи, устройство высвобождает блоки. Для этого необходимо наличие на устройстве некоторого объема свободного места. В зависимости от типа устройства либо вам самостоятельно придется зарезервировать некоторый объем свободного места, либо устройство зарезервирует его самостоятельно, причем вы этого не увидите. В любом случае, когда устройство заполняется, сборщик мусора начинает усердно трудиться, очищая некоторые блоки, чтобы увеличивался коэффициент усиления записи.

В результате скорость работы многих устройств по мере их заполнения замедляется. Насколько именно замедляется, зависит от производителя, модели, а также архитектуры устройства. Некоторые устройства рассчитаны на высокую производительность, даже если они почти заполнены, но в целом файл объемом 100 Гбайт будет обрабатываться по-разному на SSD емкостью 160 и 320 Гбайт. Замедление вызвано необходимостью при отсутствии свободных блоков ждать завершения процесса стирания. Запись в свободный блок занимает пару сотен микросекунд, но стирание происходит намного медленнее — обычно несколько миллисекунд.

Флеш-технологии

Существуют два основных типа флеш-устройств. При покупке флеш-памяти важно понимать, в чем их различие. Эти два типа являются *одноуровневой ячейкой* (single-level cell, SLC) и *многоуровневой ячейкой* (multi-level cell, MLC).

SLC хранит в ячейке один бит данных — либо 0, либо 1. SLC является относительно затратной, но она очень быстрая и долговечная — с временем жизни до 100 000 циклов записи в зависимости от производителя и модели. Это может показаться неожиданным, но на самом деле хорошее устройство SLC должно функционировать около 20 лет, и, говорят, она более долговечна и надежна, чем контроллер, в котором установлена карта. В то же время плотность хранения у нее относительно низкая, поэтому трудно обеспечить много пространства для хранения на одном устройстве.

Современные MLC хранят два бита в ячейке, однако на рынок выходят трехбитовые устройства. Тем самым достигается намного большая плотность хранения (большая емкость). Затраты у нее ниже, но то же самое относится и к скорости доступа и долговечности. Хорошее устройство MLC может быть рассчитано приблизительно на 10 000 циклов записи.

Оба типа флеш-устройств активно развиваются и конкурируют друг с другом. В настоящее время SLC по-прежнему имеет репутацию корпоративного серверного решения для хранения данных, а MLC обычно рассматривается как потребительский класс для использования в ноутбуках, камерах и т. д. Однако ситуация меняется —

недавно появился новый тип хранилища данных, так называемая корпоративная MLC (enterprise MLC, eMLC).

Развитие технологии MLS — очень интересный процесс. Ее стоит рассмотреть внимательнее, если вы задумываетесь о приобретении флеш-накопителя. MLC очень сложна, на качество и производительность устройства, использующего MLC, влияет множество факторов. Любой чип сам по себе недолговечен, у него относительно небольшой срок службы и высокая вероятность ошибок, которые необходимо исправить. По мере того как на рынке появляются еще более мелкие чипы с более высокой плотностью, ячейки которых могут хранить три бита, отдельные чипы становятся все менее надежными и более подверженными ошибкам.

Однако это не является непреодолимой инженерной проблемой. Производители конструируют устройства со все большим и большим объемом скрытых резервных мощностей. Таким образом, появляется некоторая внутренняя избыточность. Ходят слухи, что, возможно, некоторые устройства могут хранить в два раза больше данных, чем заявлено. Однако производители флеш-памяти очень тщательно охраняют свои коммерческие секреты. Еще один способ сделать MLC-чипы более прочными — изменить логику прошивки. Большое значение имеют алгоритмы контроля равномерности износа и перераспределения памяти. Следовательно, долговечность зависит от истинной емкости, логики прошивки и пр., то есть в конечном счете от производителя. Мы слышали об устройствах, уничтоженных за пару недель интенсивного использования!

В результате наиболее важными аспектами устройства MLC являются встроенные в него алгоритмы и вычислительные средства. Гораздо сложнее создать хорошее устройство MLC, чем устройство SLC, но все возможно. Самые лучшие производители, использующие отличные инженерные решения и увеличивающие емкость и плотность, предлагают устройства, достойные метки eMLC. Это определенно та область, в которой стоит внимательно отслеживать прогресс — то, что мы здесь советуем относительно MLC по сравнению с SLC, скорее всего, устареет довольно быстро.

Сколько прослужит ваше устройство

Virident гарантирует, что его устройство FlashMax 1.4 TB MLC будет работать на 15 Пбайт записей, но на уровне флеш-памяти, а видимые пользователю операции записи усилены. Мы провели небольшой эксперимент по определению коэффициента усиления записи для конкретной рабочей нагрузки.

Мы создали набор данных объемом 500 Гбайт и запустили эталонный тест tpcc-mysql на час. В течение этого времени утилита /proc/diskstats сообщила о записях объемом 984 Гбайт, а утилита конфигурирования Virident показала 1,125 Гбайт записей на уровне флеш-памяти с коэффициентом усиления записи 1,14. Помните, что он будет выше, если на устройстве используется больше места, при этом он зависит от того, являются операции записи последовательными или произвольными.

При таком коэффициенте, если проводить эталонное тестирование непрерывно в течение полутора лет, устройство окажется полностью изношенным. Конечно, большинство реальных рабочих нагрузок никогда не приближаются к такой интенсивности операций записи, поэтому карта должна использоваться много лет. Основная цель этой врезки — сказать не о том, что устройство будет быстро изнашиваться, а о том, что коэффициент усиления записи трудно предсказать: стоит проверить устройство при своей рабочей нагрузке и посмотреть, как оно себя ведет.

Как мы уже упоминали, размер также имеет большое значение для долголетия устройства. Большие устройства будут работать значительно дольше. Поскольку в настоящее время все чаще применяются довольно большие мощности, MLC становится все более популярной, и причина этого заключается в ее долговечности.

Эталонное тестирование флеш-памяти

Эталонное тестирование флеш-памяти — процесс довольно сложный. Есть множество способов сделать это неправильно, а чтобы сделать правильно, нужно обладать специфическими знаниями о конкретном устройстве, а также проявить внимательность и терпение.

У флеш-устройств есть три стадии, которые мы назовем характеристиками производительности: A, B и C. Они быстро приступают к работе (этап A), а затем начинает работать сборщик мусора. После этого наступают период перехода в устойчивое состояние (этап B) и, наконец, само устойчивое состояние (этап C). У всех протестированных нами устройств наблюдался такой порядок действий.

Конечно, вас интересует производительность на этапе C, поэтому эталонные тесты должны измерять только эту часть прогона. Это означает, что эталонный тест должен быть чем-то большим, чем просто эталонный тест: он должен состоять из рабочей нагрузки для «прогрева» и собственно эталонного теста. Тем не менее определить, где заканчивается «прогревочная» нагрузка и начинается собственно эталонный тест, может быть довольно сложно.

Устройства, файловые системы и операционные системы по-разному поддерживают команду `TRIM`, которая обозначает пространство как готовое к повторному использованию. Иногда устройство запускает команду `TRIM` при удалении всех файлов. Если это произойдет между прогонами эталонного теста, устройство перейдет на этап A и вам придется циклически проходить через этапы A и B между прогонами. Другим фактором является изменение производительности, когда устройство более или менее заполнено. Повторяемый эталонный тест должен учитывать все эти факторы.

Все эти сложности делают эталонные тесты и спецификации от поставщиков минным полем для невнимательного пользователя, даже если он все выполняет добро-

совестно. Обычно вы получаете от поставщиков четыре числа. Приведем пример характеристик устройства.

1. Устройство может считывать со скоростью до 520 Мбайт/с.
2. Устройство может записывать со скоростью до 480 Мбайт/с.
3. Устройство может выполнять длительную запись со скоростью до 420 Мбайт/с.
4. Устройство за 1 секунду может выполнять 70 000 произвольных операций записи размером 4 Кбайт.

Если вы сравните эти числа, то заметите, что максимальные операции ввода/вывода в секунду (input/output operations per second, IOPS) 70 000 произвольных записей размером 4 Кбайт в секунду составляют всего около 274 Мбайт/с, что намного меньше, чем максимальная пропускная способность записи, упомянутая в пунктах 2 и 3. Это связано с тем, что максимальная пропускная способность записи достигается при больших размерах блоков — 64 или 128 Кбайт, а максимальный IOPS достигается при небольших размерах блоков.

Большинство приложений не могут записывать такими большими блоками. InnoDB обычно записывает комбинацию блоков размером 16 Кбайт и 512-байтовых блоков. В результате от этого устройства можно ожидать скорости записи только 274 Мбайт/с, и это на этапе A, прежде чем сборщик мусора начнет работать и устройство выйдет на стабильный долгосрочный уровень производительности!

Современные эталонные тесты MySQL и файлы нагрузки операций ввода/вывода на твердотельные устройства можно найти в наших блогах по адресам: http://www.ssdperformanceblog.com, http://www.mysqlperfor.manceblog.com.

Твердотельные диски

SSD эмулируют жесткие диски SATA. Эти возможности совместимы: замена накопителя SATA не требует каких-либо специальных драйверов или соединений.

Диски Intel X-25E, вероятно, являются наиболее распространенными SSD, которые используются сегодня на серверах, но существуют и другие варианты. X-25E продается на «корпоративном» рынке, но есть также X-25M, который имеет хранилище MLC и предназначен для массового рынка пользователей ноутбуков и пр. Intel также продает серию 320, которую используют множество людей. И это всего лишь один производитель, а их много, и к тому моменту, когда эта книга выйдет в печать, что-то из того, что мы написали о SSD, скорее всего, уже устареет.

Хорошая новость о SSD заключается в том, что они легкодоступны, относительно дешевы и работают намного быстрее, чем жесткие диски. К тому же существует множество брендов и моделей. Самый большой недостаток заключается в том, что они не всегда так надежны, как жесткие диски, причем надежность различается в зависимости от марки и модели. До недавнего времени у большинства устройств не было встроенной батареи, но был кэш записи для буферизации. Этот кэш ненадежен без поддерживающей его батареи, но он не может быть отключен без значительного увеличения нагрузки на запись на базовом флеш-накопителе. Таким образом, если вы

отключите кэш своего накопителя, чтобы получить действительно надежное хранилище, устройство будет изнашиваться быстрее, а в некоторых случаях это к тому же аннулирует гарантию.

Некоторые производители не спешат сообщать людям об этой особенности их SSD, кроме того, они ревниво охраняют такие детали, как внутренняя архитектура устройств. Вопрос о том, есть ли у них батарея или конденсатор, которые могут обеспечить безопасность данных кэша записи в случае сбоя питания, обычно остается без ответа. В некоторых случаях диск будет принимать команду отключения кэша, но игнорировать ее. Таким образом, вы действительно не будете знать, долговечен ли ваш накопитель, пока не проведете краш-тест. Мы провели краш-тест некоторых дисков и получили разные результаты. Сейчас некоторые диски поставляются с конденсатором для защиты кэша, что увеличивает их долговечность, но в целом, если производители вашего диска не хвастаются, что у него есть батарея или конденсатор, то их и нет. Это означает, что в случае сбоев питания он прослужит недолго, поэтому ваши данные будут повреждены, а вы, возможно, даже не узнаете об этом. Конденсатор или батарея — что-то обязательно должно быть в вашем SSD.

Обычно, приобретая SSD, вы получаете то, за что платите. Проблемы базовой технологии решить нелегко. Многие производители делают диски, которые быстро приходят в негодность под нагрузкой или не обеспечивают стабильной работы. У некоторых производителей низкокачественного оборудования есть манера постоянно выпускать новые поколения дисков, утверждая при этом, что они решили все проблемы предыдущего поколения. Конечно, это не так. Если вы заботитесь о надежности и стабильно высокой производительности, приобретайте устройства корпоративного класса — обычно они стоят денег, которые за них придется заплатить.

Использование RAID с SSD. Мы рекомендуем использовать RAID (redundant array of inexpensive disks — избыточный массив независимых дисков) с SSD-накопителями SATA. Один диск просто недостаточно надежен, чтобы доверять ему ваши данные.

Многие старые RAID-контроллеры не были готовы к работе с SSD. Предполагалось, что они управляют жесткими дисками на шпинделе и выполняют буферизацию и записи с перестановкой, полагая, что так будет эффективнее. Это была просто лишняя работа и дополнительная задержка, поскольку логическое местоположение, которое предоставляет SSD, сопоставляется с произвольным местоположением в базовой флеш-памяти. Сейчас ситуация немного улучшилась. На конце номеров моделей некоторых RAID-контроллеров указывается буква, обозначающая, что они работают с SSD. Например, у контроллеров Adaptec для этой цели используется буква Z.

Тем не менее даже готовые к работе с флеш-устройствами контроллеры на самом деле таковыми не являются. Вадим провел эталонное тестирование контроллера Adaptec 5805Z с различными дисками в RAID 10, используя файл объемом 500 Гбайт и конкурентность 16. Результаты были ужасными: 95-я процентильная задержка для произвольных записей находилась в двузначных миллисекундах, а в худшем случае составила более секунды[1] (приемлемый результат должен быть меньше 1 миллисекунда).

[1] Но это еще не все. Мы проверили диски после эталонного тестирования и обнаружили два «убитых» SSD и один с повреждениями.

Мы выполняли специальное исследование для клиента, который хотел посмотреть, будут ли SSD Micron лучше 64-гигабайтовых Intel SSD, которые они уже использовали в той же конфигурации. Проведя эталонное тестирование дисков Intel, мы получили те же характеристики производительности. Поэтому попробовали некоторые другие конфигурации накопителей, с SAS-экспандером и без него. Результаты приведены в табл. 9.1.

Таблица 9.1. Эталонные тесты SSD в RAID-контроллере Adaptec

Диск	Марка	Размер, Гбайт	SAS-экспандер	Произвольное чтение, Мбайт/с	Произвольная запись, Мбайт/с
34	Intel	64	Да	310	130
14	Intel	64	Да	305	145
24	Micron	50	Нет	350	120
34	Intel	50	Нет	350	180

Ни один из этих результатов даже близко не стоял к нашим ожиданиям от использования стольких дисков. В целом RAID-контроллер давал производительность, которой мы ожидали от шести или восьми дисков, а не от десяти. RAID-контроллер был просто насыщен. Суть этой истории в том, что следует проводить тщательные эталонные тесты, прежде чем вкладывать значительные ресурсы в аппаратные средства — результаты могут сильно отличаться от ваших ожиданий.

Устройства хранения PCIe

В отличие от SATA SSD устройства PCIe не пытаются эмулировать жесткие диски. Это хорошо: интерфейс между сервером и жесткими дисками не справляется с флеш-памятью, когда она работает на полную мощность. Соединение SAS/SATA имеет более низкую пропускную способность, чем PCIe, поэтому с точки зрения производительности лучше выбрать PCIe. Кроме того, у устройств PCIe гораздо более низкая задержка, поскольку они физически ближе к процессорам.

Никакие другие устройства не могут дать такой же производительности, как устройства PCIe. Недостатком же является их относительная дороговизна.

Всем известным нам моделям требуются специальные драйверы для создания блочного устройства, которое операционная система видит как жесткий диск. Они используют сочетание стратегий контроля равномерности износа и других, некоторые из них используют центральный процессор и память хост-системы, а некоторые обладают встроенными логическими контроллерами и оперативной памятью. Во многих случаях хост-система имеет множество ресурсов ЦП и ОЗУ, поэтому использование их на самом деле является более рентабельным, чем покупка карты с собственной оперативной памятью.

Мы не советуем использовать RAID с устройствами PCIe. Они слишком дороги для использования с RAID, к тому же у большинства устройств есть собственный RAID-массив. Мы не знаем, насколько вероятно то, что контроллер будет выходить

из строя, но поставщики утверждают, что их контроллеры в целом так же хороши, как и сетевые карты или RAID-контроллеры, и это, наверное, окажется правдой. Другими словами, среднее время между отказами (MTBF) для этих устройств, скорее всего, будет аналогично данной характеристике материнской платы, поэтому использование вместе с ними RAID просто увеличит затраты, но не принесет большой выгоды.

Флеш-карты PCIe выпускают несколько производителей. Наиболее популярными брендами у пользователей MySQL являются Fusion-io и Virident, есть неплохие предложения также у Texas Memory Systems, STEC и OCZ. Доступны как SLC-, так и MLC-карты.

Другие типы твердотельных хранилищ данных

Помимо устройств SSD и PCIe, существуют другие варианты от таких компаний, как Violin Memory, SandForce и Texas Memory Systems. Эти компании выпускают большие устройства с флеш-памятью, которые по существу являются флеш-SAN с десятками терабайт памяти. Используют их в основном для крупномасштабной консолидации хранилищ данных. Они очень дорогие и очень высокопроизводительные. Нам известны несколько человек, которые используют их, и несколько раз мы анализировали работу этих устройств. Они обеспечивают очень приличную задержку независимо от времени запроса по сети, например, менее 4 миллисекунд задержки при использовании NFS.

Тем не менее эти хранилища данных не очень хорошо подходят для обычного рынка MySQL. Они предназначены скорее для других баз данных, таких как Oracle, которые могут использовать их для кластеризации с общим хранилищем. В целом MySQL не может воспользоваться таким мощным хранилищем в таких больших масштабах, поскольку оно плохо работает с базами данных объемом в десятки терабайт. Ответом MySQL на такую большую базу данных является шардирование или горизонтальное масштабирование в архитектурах без шардирования.

Однако эти большие запоминающие устройства могут использоваться в специализированных решениях, например Infobright. ScaleDB может быть развернута в архитектуре с общим хранилищем, но мы еще не видели ее в реальных условиях, поэтому не знаем, как хорошо она будет работать.

Когда стоит использовать флеш-устройства

Наиболее очевидным вариантом использования твердотельного хранилища данных является любая рабочая нагрузка с большим количеством произвольных операций ввода/вывода. Произвольный ввод/вывод обычно связан с тем, что данные больше, чем память сервера. При использовании стандартных жестких дисков вы ограничены скоростью вращения и задержкой поиска. Флеш-устройства могут значительно улучшить ситуацию.

Конечно, в некоторых случаях можно купить больше оперативной памяти, чтобы ее хватало для произвольной рабочей нагрузки, а операции ввода/вывода просто исчезли. Но когда такой возможности нет, флеш-устройства могут сильно помочь. Еще одна проблема, которую далеко не всегда можно решить с помощью ОЗУ, — это рабочая нагрузка на запись с высокой пропускной способностью. Добавление памяти поможет уменьшить нагрузку на запись, которая обращается к дискам, поскольку увеличение памяти создает больше возможностей для буферизации и комбинирования записей. Это позволяет преобразовать рабочую нагрузку с произвольной записью в более последовательную.

Однако этот прием нельзя использовать до бесконечности, и некоторые транзакционные или рабочие нагрузки с большим количеством операций вставки не всегда выигрывают от его применения. В данном случае также может пригодиться флеш-хранилище.

Однопоточные рабочие нагрузки — еще один характерный сценарий, при котором флеш-устройства потенциально могут улучшить ситуацию. Когда рабочая нагрузка является однопоточной, она очень чувствительна к задержке, и более низкая задержка твердотельного хранилища будет иметь большое значение. Напротив, многопоточные рабочие нагрузки часто легко распараллеливаются, что приводит к увеличению пропускной способности. Репликация MySQL является характерным примером однопоточной рабочей нагрузки, которая получает множество преимуществ из-за снижения задержки. Использование флеш-устройства на подчиненных серверах часто может значительно повысить их производительность, когда они начинают отставать от главного сервера.

Флеш-устройства также отлично подходят для консолидации серверов, особенно в формфакторе PCIe. Нам доводилось видеть консолидацию нескольких серверов на одном физическом — иногда количество виртуальных серверов доходило до 10 и даже 15 (см. главу 11 для получения дополнительной информации по этой теме).

Однако флеш-устройства хороши не всегда. Хороший пример — их использование для рабочей нагрузки с последовательными записями, такими как файлы журнала InnoDB. В этих условиях флеш-устройства не дают значительного выигрыша в соотношении затрат и производительности, поскольку при последовательной записи работают не намного быстрее, чем стандартные жесткие диски. Такая рабочая нагрузка также характеризуется высокой пропускной способностью, от которой устройство быстрее изнашивается. Зачастую лучше хранить ваши файлы журналов на стандартных жестких дисках с помощью RAID-контроллера с кэшем записи с автономным питанием.

Иногда выбор устройства зависит от соотношения «память — диск», а не только от диска. Если вы можете купить достаточное количество ОЗУ для кэширования своей рабочей нагрузки, это может оказаться дешевле и эффективнее, чем покупка устройства флеш-памяти.

Использование технологии Flashcache

Хотя существует множество возможностей нахождения компромисса между флеш-памятью, жесткими дисками и оперативной памятью, совсем не обязательно рассматривать их как однокомпонентные уровни иерархии хранения. Иногда имеет смысл использовать комбинацию дисковых технологий и памяти, и этим занимается модуль Flashcache.

Технология Flashcache — это одна из возможных реализаций методики, которую можно найти во множестве систем, например в базе данных Oracle, файловой системе ZFS и даже во многих современных жестких дисках и RAID-контроллерах. Многое из того, что будет сказано в дальнейшем, подходит к большому количеству систем, но мы сосредоточимся только на технологии Flashcache, поскольку она не зависит от производителя и файловой системы

Модуль Flashcache — это модуль ядра Linux, в котором используется устройство отображения Linux. Он создает в иерархии памяти промежуточный уровень между ОЗУ и диском. Это одна из технологий с открытым исходным кодом, созданная Facebook, которая используется для оптимизации аппаратного обеспечения Facebook при рабочей нагрузке, связанной с базой данных.

Модуль Flashcache создает блочное устройство, которое можно разделить на секции и использовать для создания файловой системы, как и любое другое. Хитрость заключается в том, что это блочное устройство поддерживается как флеш-памятью, так и дисковой памятью. Флеш-устройство используется как интеллектуальный кэш для чтения и записи.

Виртуальное блочное устройство намного больше, чем флеш-устройство, но это нормально, потому что диск считается окончательным репозиторием для данных. Флеш-устройство необходимо для буферизации записи и эффективного расширения памяти сервера для кэширования операций чтения.

Насколько же высока будет производительность? Похоже, что у модуля Flashcache относительно высокие накладные затраты ядра. (Модуль отображения устройств, похоже, не так эффективен, как должен быть, но мы не исследовали причины этого.) Но хотя кажется, что модуль Flashcache теоретически может быть более эффективным, а его максимальная производительность меньше, чем производительность базового флеш-накопителя, он все же намного быстрее, чем диски. Поэтому возможность его использования, по-видимому, стоит рассмотреть.

Мы оценивали производительность модуля Flashcache с помощью нескольких сотен эталонных тестов и обнаружили, что довольно сложно осмысленно тестировать искусственную рабочую нагрузку. Мы не сумели понять, насколько полезен модуль Flashcache для рабочих нагрузок, связанных с операциями записи в целом, но для рабочей нагрузки, связанной с чтением, он может быть весьма хорош. Это соответствует варианту использования, для которого он был разработан: для серверов, нагрузка которых состоит в основном из операций чтения, а рабочее множество гораздо больше, чем размер памяти.

Помимо лабораторных испытаний, у нас есть некоторый опыт работы с Flashcache при рабочих нагрузках. Вспоминается случай с четырехтерабайтной базой данных. Она сильно пострадала от задержки репликации. Мы модифицировали систему, добавив карту Virident PCIe с объемом памяти 0,5 Тбайт. Затем установили Flashcache и использовали карту PCIe как флеш-часть устройства. Это удвоило скорость репликации.

Использование технологии Flashcache наиболее экономично, когда флеш-карта почти заполнена, поэтому стоит иметь карту, производительность которой не сильно ухудшается при ее заполнении. Именно поэтому мы выбрали карту Virident.

Модуль Flashcache — это кэш, поэтому он должен прогреться так же, как и любой другой. Хотя период прогрева может быть очень длинным. Например, в случае, о котором мы только что упоминали, модулю Flashcache потребовалась неделя, чтобы разогреться и действительно ускорить производительность.

Стоит ли использовать Flashcache? Всякое может быть. Поэтому мы считаем, что если вы не уверены в правильности такого решения, стоит проконсультироваться с экспертами. Сложно понять механику технологии Flashcache и то, как она влияет на размер рабочего множества базы данных и на три уровня (по меньшей мере) хранилища под базой данных.

- Прежде всего, есть буферный пул InnoDB, отношение размера которого к размеру рабочего множества определяет один из показателей непопадания в кэш. Извлечение данных из этого кэша происходит очень быстро, а время отклика очень равномерное.
- Непопадания из буферного пула распространяются до модуля Flashcache, который имеет сложное распределение времени отклика. Уровень непопадания в кэш для модуля Flashcache определяется размером рабочего множества и размером флеш-устройства, которое его поддерживает. Извлечение данных из этого кэша происходит намного быстрее, чем с диска.
- Непопадания в кэш модуля Flashcache распространяются вплоть до дисков, которые имеют довольно равномерное распределение времени отклика.

Под этими уровнями могут быть и другие: ваши SAN-сети или кэш RAID-контроллера, например.

Проведем мысленный эксперимент, который иллюстрирует взаимодействие этих слоев. Очевидно, что время отклика от модуля Flashcache не будет таким же стабильным или небольшим, как было бы от одного флеш-устройства. Но представьте, что у вас есть 1 Тбайт данных и 99 % операций ввода/вывода на протяжении длительного периода связаны со 100 Гбайт из этих данных. То есть в течение долгого времени 99-й процентиль рабочего множества составляет 100 Гбайт.

Предположим теперь, что у нас есть следующие устройства хранения данных: большой том RAID, который может выполнять 1000 IOPS, и гораздо меньшее флеш-устройство, которое может выполнять 100 000 IOPS. Флеш-устройство — допустим, емкостью 128 Гбайт — недостаточно велико, чтобы хранить все данные

поэтому использование его одного нас не спасет. Если мы будем использовать флеш-устройство для модуля Flashcache, можно будет ожидать, что извлечение из кэша будут выполняться намного быстрее, чем извлечение с диска, но медленнее, чем отклики самого устройства флеш-памяти. Давайте немного округлим и скажем, что 90 % запросов на модуль Flashcache можно обслуживать со скоростью, эквивалентной 50 000 IOPS.

Каковы же результаты этого мысленного эксперимента? Есть два основных момента.

1. Наша система обеспечивает намного лучшую производительность с помощью технологии Flashcache, чем без нее, поскольку большинство непопаданий в кэш буферного пула передаются на флеш-карту и обслуживаются с намного большей скоростью чем извлечение с диска (99-й процентиль рабочего множества полностью помещается на флеш-карту).
2. Девяносто процентов попаданий в кэш на модуле Flashcache означает, что уровень непопаданий — 10 %. Поскольку базовые диски могут обслуживать только 1000 IOPS, самое большее, чего можно ожидать от модуля Flashcache, — 10 000 IOPS. Чтобы понять, почему это так, представьте, что произойдет, если мы потребуем больше операций: при 10 % непопаданий в кэш операций ввода/вывода и переходе этих запросов на том RAID мы будем запрашивать более 1000 IOPS от тома RAID, а мы знаем, что с этой нагрузкой он справиться не может. В результате, хотя модуль Flashcache работает медленнее, чем флеш-карта, система в целом по-прежнему ограничена скоростью работы тома RAID, а не флеш-карты или модуля Flashcache.

Таким образом, вопрос о том, подходит ли вам технология Flashcache, сложен, а ответ на него зависит от множества факторов. В общем случае кажется, что он лучше всего подходит для рабочих нагрузок с множеством операций ввода/вывода, большая часть которых — это операции чтения, при этом экономично использовать память не представлялось бы возможным из-за слишком большого размера рабочего множества.

Оптимизация MySQL для твердотельных хранилищ данных

Если вы используете MySQL на флеш-устройстве, то можете применить некоторые параметры конфигурации для обеспечения лучшей производительности. В частности, конфигурация InnoDB по умолчанию связана с жесткими дисками, а не с твердотельными хранилищами. Не все версии InnoDB обеспечивают одинаковый уровень конфигурируемости. Многие улучшения, разработанные для флеш-устройств, появились на Percona Server, хотя часть из них уже были реализованы в версии InnoDB от Oracle или, возможно, планируются для будущих версий. Улучшения включают в себя следующие возможности.

- *Увеличение мощности ввода/вывода InnoDB.* Флеш-устройства поддерживают гораздо более высокий уровень конкурентности, чем обычные жесткие диски, поэтому можно увеличить количество потоков ввода/вывода для чтения и за-

писи до 10 или 15, что не ухудшит результаты. Кроме того, можно увеличить значение параметра `innodb_io_capacity` до 2000–20 000 в зависимости от IOPS, которые может выполнить ваше устройство. Это особенно актуально для официального InnoDB от Oracle, у которой много внутренних алгоритмов, зависящих от этого параметра.

- *Увеличение размера файлов журнала InnoDB.* Даже при улучшенных алгоритмах восстановления, используемых в последних версиях InnoDB, вы не захотите, чтобы файлы журналов на жестких дисках были слишком большими, поскольку произвольный ввод/вывод, необходимый для восстановления после сбоя, выполняется медленно и может привести к значительному увеличению времени восстановления. Флеш-память позволяет сделать это намного быстрее, поэтому вы сможете иметь большие файлы журнала InnoDB, что приведет к улучшению и стабилизации производительности. Это особенно необходимо для официального InnoDB от Oracle, у которой есть проблемы с поддержанием постоянного уровня сброса «грязных» страниц, если файлы журналов недостаточно велики — на момент написания книги 4 Гбайт или больше кажутся нормой для типичных серверов. Percona Server и MySQL 5.6 поддерживают файлы журналов размером более 4 Гбайт.
- *Перемещение некоторых файлов с флеш-памяти на RAID.* Помимо того, что файлы журнала InnoDB больше, может быть целесообразно хранить файлы журнала отдельно от файлов данных, размещая их в RAID-контроллере с кэшем записи с автономным питанием, а не на твердотельном устройстве. На это есть несколько причин. Одна из них заключается в том, что тип ввода/вывода, с которым сталкиваются файлы журнала, ненамного быстрее на флеш-устройствах, чем на настроенном таким образом RAID. InnoDB записывает файлы журнала последовательно в 512-байтовые блоки и не читает их никогда, кроме как во время восстановления после сбоя, когда она делает это последовательно. Для таких операций бесполезно использовать флеш-память. Также неплохо переместить эти небольшие записи в том RAID, потому что очень маленькие записи увеличивают коэффициент усиления записи на флеш-устройствах, что может вызвать проблемы с долговечностью некоторых из них. Кроме того, смесь записей большого и маленького объема может вызвать повышенную задержку на некоторых устройствах.

  По схожим причинам иногда полезно перемещать в том RAID файлы двоичного журнала, возможно, стоит переместить и файл `ibdata1`. Файл `ibdata1` содержит буфер двойной записи и буфер вставки. Буфер двойной записи, в частности, получает много повторных записей. В Percona Server вы можете удалить буфер двойной записи из файла `ibdata1` и сохранить его в отдельном файле, который можно разместить в томе RAID.

  Есть еще один вариант: вы можете использовать Percona Server для записи журналов транзакций в четырехкилобайтовых блоках вместо 512-байтовых блоков. Это может оказаться более эффективным с точки зрения как флеш-памяти, так и самого сервера.

  Все приведенные рекомендации по большей части специфичны для конкретного оборудования, и вам они могут подходить не в полной мере, поэтому убедитесь,

что понимаете связанные с ними факторы, и проверяйте их соответствующим образом, прежде чем принимать важные решения по изменению формы хранилища данных.

- *Отключение упреждающего чтения.* Упреждающее чтение оптимизирует доступ к устройствам, делая прогноз по поводу шаблонов чтения и запрашивая данные с устройства, если есть основания считать, что они понадобятся в будущем. На самом деле в InnoDB есть два типа упреждающего чтения, и при различных обстоятельствах мы обнаружили, что проблемы с производительностью могут быть вызваны как собственно упреждающим чтением, так и тем, как оно работает. Накладные затраты во многих случаях больше, чем выигрыш, особенно на флеш-накопителе, однако у нас нет убедительных доказательств или рекомендаций относительно того, насколько вы можете повысить производительность, отключив упреждающее чтение.

  Oracle отключил так называемое произвольное упреждающее чтение в плагине InnoDB в MySQL 5.1, а затем повторно включил в MySQL 5.5, снабдив параметром для его настройки. Percona Server позволяет настраивать как произвольное, так и последовательное упреждающее чтение и в старых версиях сервера.

- *Конфигурирование алгоритма сброса InnoDB.* Способ, которым InnoDB решает, когда, сколько и какие страницы сбросить, — тема очень сложная для изучения, и у нас здесь нет возможности ее подробно обсудить. Кроме того, эта проблема является предметом активных исследований, и на самом деле в различных версиях InnoDB и MySQL используются разные алгоритмы.

  Стандартные алгоритмы InnoDB не предполагают наличия широких конфигурационных возможностей, которые позволяют лучше использовать флеш-память, но если вы применяете Percona XtraDB (входит в Percona Server и MariaDB), рекомендуем установить для параметра `innodb_adaptive_checkpoint` значение `keep_average` вместо значения по умолчанию `estimate`. Это поможет обеспечить более стабильную работу и избежать серверных стопоров, поскольку алгоритм `estimate` может застопориться на флеш-памяти. Мы разработали режим `keep_average` специально для флеш-памяти, поскольку понимали, что можно передавать на устройство столько операций ввода/вывода, сколько хочется, не создавая узкого места и последующего стопора.

  Кроме того, мы рекомендуем устанавливать для параметра `innodb_flush_neighbor_pages` значение `0` на флеш-памяти. Это не позволит InnoDB найти соседние «грязные» страницы, чтобы сбросить их вместе. Алгоритм, который выполняет такую операцию, может вызвать сильный всплеск записи, высокую задержку и внутреннюю конкуренцию. Это не обязательно и не полезно для флеш-памяти, потому что соседние страницы можно сбрасывать отдельно, не влияя на производительность.

- *Потенциальное отключение буфера двойной записи.* Вместо перемещения буфера двойной записи с флеш-устройства вы можете полностью отключить его. Некоторые производители утверждают, что их устройства поддерживают неделимую запись объемом 16 Кбайт, что делает буфер двойной записи избыточным. Необходимо убедиться, что вся система хранения настроена на поддержку неделимых

операций объемом 16 Кбайт. Обычно для этого требуются `O_DIRECT` и файловая система XFS.

У нас нет убедительных доказательств того, что утверждение о неделимости верно, но знание принципов работы флеш-памяти позволяет считать, что вероятность частичной записи страниц в файлы данных значительно снижается. При этом выигрыш на флеш-устройствах намного больше, чем на обычных жестких дисках. Отключение буфера двойной записи может улучшить общую производительность MySQL на флеш-памяти примерно на 50 %, поэтому, хотя мы и не уверены, что это на 100 % безопасно, об этом стоит подумать.

- *Ограничение размера буфера вставки.* Буфер вставки (или буфер изменений в новых версиях InnoDB) предназначен для уменьшения количества операций произвольного ввода/вывода страниц неуникальных вторичных индексов, которые при обновлении строк не находятся в памяти. На жестких дисках он может существенно снизить количество операций произвольного ввода/вывода. Для некоторых рабочих нагрузок в случае, когда рабочее множество намного больше, чем объем памяти, разница может достигать почти двух порядков. В таких случаях очень полезно увеличить размер буфера вставки.

  Однако для флеш-памяти это не так важно. Произвольный ввод/вывод на флеш-устройствах выполняется намного быстрее, поэтому даже полное отключение буфера вставки не принесет больших проблем. Однако вы вряд ли будете его отключать. Лучше оставить его включенным, поскольку операции ввода/вывода — это только часть всех затрат на обновление страниц индекса, которые не находятся в памяти. Основной параметр, который необходимо настроить на флеш-устройствах, — это максимально допустимый размер буфера вставки. Можно установить сравнительно небольшую верхнюю границу, тем самым не разрешать ему разрастаться. Это позволит избежать использования большого количества места на вашем устройстве и предотвратит увеличение размера файла `ibdata1`.

Помимо вышеупомянутых рекомендаций по конфигурации, разные эксперты предлагали и некоторые другие способы оптимизации для флеш-памяти. Однако с ними все далеко не очевидно, поэтому мы их только назовем и предоставим вам самостоятельно оценить их преимущества в каждом конкретном случае. Прежде всего это размер страницы InnoDB. Мы получали противоречивые результаты, поэтому не можем дать каких-либо конкретных рекомендаций. Хорошо, что в Percona Server размер страницы настраивается без перекомпиляции сервера, и это будет возможно также в MySQL 5.6. Предыдущие версии MySQL требовали перекомпиляции сервера при изменении размера страницы, основная масса пользователей работала со стандартными страницами размером 16 Кбайт. Если изменять размер страницы станет проще, можно будет ожидать гораздо большего числа экспериментов с нестандартными размерами. Вполне вероятно, что эти многочисленные эксперименты дадут нам массу полезной информации.

Другим способом оптимизации является применение альтернативных алгоритмов для контрольных сумм страницы InnoDB. Если система хранения реагирует очень

быстро, время вычисления контрольной суммы фактически может оказаться значительным по сравнению с временем, затрачиваемым на операцию ввода/вывода. В некоторых случаях это, а не операции ввода/вывода может стать узким местом. Наши эталонные тесты не показали результатов, которые можно было бы применить к широкому спектру вариантов использования, поэтому ничего определенного посоветовать не можем. Percona XtraDB позволяет вам изменить алгоритм расчета контрольной суммы. Такая же возможность будет реализована и в MySQL 5.6.

Возможно, вы обратили внимание на то, что мы много говорили о возможностях оптимизации, которые пока недоступны в стандартной InnoDB. Надеемся и верим, что многие улучшения, которые мы реализовали в Percona Server и XtraDB, в конечном итоге станут доступны для более широкой аудитории. Тем не менее, даже если вы используете официальный дистрибутив MySQL от Oracle, существуют мероприятия, которые вы можете предпринять для оптимизации работы своего сервера с флеш-памятью. Вам нужно использовать `innodb_file_per_table` и поместить каталог данных на флеш-устройство. А затем переместить файлы журнала и `ibdata1`, а также все остальные файлы журналов (двоичные журналы, журналы ретрансляции и т. д.) в том RAID, как мы писали ранее. Это сконцентрирует нагрузку произвольного ввода/вывода на флеш-устройстве и по максимуму перенесет с этого него операции последовательной файловой записи. Тем самым на флеш-устройстве будет сэкономлено место, а его износ уменьшится.

Помимо этого при любых версиях сервера необходимо убедиться, что включена гиперпоточность. Она хорошо помогает при использовании флеш-памяти, поскольку диск, как правило, больше не является узким местом, а рабочая нагрузка становится сильнее зависимой от процессора, а не от операций ввода/вывода.

Выбор оборудования для подчиненного сервера

При выборе оборудования для подчиненного сервера руководствуются в основном теми же принципами, что и при выборе оборудования для главного сервера, хотя есть и некоторые отличия. Если вы планируете использовать подчиненный сервер, чтобы переключаться на него в случае сбоя основного, то, как правило, он должен быть не менее мощным, чем главный сервер. И вне зависимости от того, будет ли подчиненный сервер выступать в роли резервной замены главного, он должен быть достаточно мощным для выполнения всех операций записи, произведенных на главном сервере, с учетом того, что они должны еще и выполняться последовательно (дополнительная информация об этом приведена в главе 10).

Основное соображение при выборе оборудования для подчиненного сервера — затраты: нужно ли тратить на него столько же средств, сколько и на главный? Можно ли сконфигурировать подчиненный сервер по-другому, чтобы выжать из него дополнительную производительность? Будет ли рабочая нагрузка подчиненного сервера отличаться от нагрузки главного? Каков возможный выигрыш от другого оборудования?

Все зависит от обстоятельств. Если подчиненный сервер используется в качестве резервного, то оборудование и конфигурацию главного и подчиненного серверов имеет смысл делать одинаковыми. Но если единственное назначение репликации — увеличить пропускную способность системы в части операций чтения, то возможны самые разные компромиссы. Например, можно использовать на подчиненном сервере другую подсистему хранения, оборудовать его более дешевыми компонентами или сконфигурировать RAID-массив уровня 0, а не 5 или 10. Кроме того, можно пожертвовать некоторыми гарантиями непротиворечивости и долговечности, чтобы подчиненному серверу приходилось меньше работать.

Эти меры могут быть экономически эффективными в больших масштабах, но при малых лишь усложняют работу. Кажется, что на практике большинство людей используют для реплик одну из двух стратегий: устанавливают везде одинаковое оборудование или покупают новое оборудование для главного сервера, а старое передают подчиненному.

Использование твердотельных дисков на подчиненном сервере может иметь смысл, когда ему нелегко идти в ногу с главным сервером. Быстрые операции произвольного ввода/вывода помогают компенсировать недостатки однопоточной репликации.

Оптимизация производительности с помощью RAID

Подсистемы хранения часто размещают все данные и индексы в одном большом файле, а это означает, что для хранения данных наиболее подходящим вариантом является технология RAID[1] (redundant array of inexpensive disks — избыточный массив независимых дисков). Она может решить проблемы резервирования, емкости, кэширования и быстродействия. Но как и для остальных рассматриваемых способов оптимизации, RAID-массив можно конфигурировать по-разному и важно выбрать уровень, отвечающий именно вашим потребностям.

Мы не станем здесь ни разбирать все уровни RAID, ни вдаваться в детали того, как организовано хранение данных на каждом из них. Эта тема прекрасно изложена в различных книгах и в Сети[2]. Мы же займемся тем, как различные конфигурации

[1] Неплохой альтернативой является также секционирование (см. главу 7), поскольку оно позволяет разбить один большой файл на много более мелких, которые можно разместить на разных устройствах. Однако при сверхбольших объемах данных RAID оказывается более простым решением по сравнению с секционированием. Пользователю не нужно балансировать нагрузку вручную или вмешиваться, когда распределение нагрузки изменяется. Кроме того, эта технология обеспечивает избыточность, которую не получить за счет разнесения секций по разным дискам.

[2] Упомянем лишь два хороших источника информации о RAID: статью в «Википедии» (http://en.wikipedia.org/wiki/RAID) и учебное руководство AC&NC, расположенное по адресу http://www.acnc.com/04_00.html.

RAID удовлетворяют требованиям, предъявляемым серверами базы данных. Далее перечислены наиболее важные уровни RAID.

- *RAID 0.* Уровень RAID 0 — самая малозатратная и наиболее производительная конфигурация RAID, по крайней мере при упрощенном подходе к оценке затрат и производительности (если рассматривать также восстановление данных, то затраты начинают расти). Поскольку этот уровень не обеспечивает никакой избыточности, мы рекомендуем его лишь для серверов, которые вам более или менее безразличны, например для подчиненных серверов или серверов, которые по той или иной причине считаются одноразовыми. Типичный пример — подчиненный сервер, который легко клонировать из другого подчиненного сервера.

  Еще раз подчеркнем, что уровень RAID 0 не обеспечивает никакой избыточности, несмотря на то что в акрониме RAID первая буква означает redundant («избыточный»). На самом деле вероятность выхода из строя RAID-массива уровня 0 даже выше, а не ниже вероятности отказа одиночного диска!

- *RAID 1.* Уровень RAID 1 обеспечивает неплохую производительность чтения во многих ситуациях, а так как данные дублируются, то имеется и избыточность. При операциях чтения RAID 1 чуть быстрее, чем RAID 0. Он хорошо подходит для серверов, занятых журналированием и другими аналогичными операциями, поскольку в случае последовательной записи редко бывает необходимо, чтобы все составляющие массив диски показывали высокое быстродействие (в отличие от произвольной записи, для которой распараллеливание может дать заметный эффект). Этот уровень также часто выбирают для маломощных серверов, которым нужна избыточность, но жестких дисков всего два.

  Уровни RAID 0 и RAID 1 очень просты и часто реализуются программно. Многие операционные системы предоставляют простые средства для создания томов типа RAID 0 или RAID 1.

- *RAID 5.* Уровень RAID 5 выглядит более сложным, но для некоторых приложений это единственный выход из-за ценовых параметров и/или ограничений на количество дисков, которые физически можно разместить на сервере. В этом случае данные распределяются по нескольким дискам и дополнительно поддерживаются распределенные блоки с контрольной суммой, так что при отказе одного диска хранившаяся на нем информация может быть восстановлена по данным на других дисках и контрольным блокам. Однако при выходе из строя двух дисков весь том становится невосстанавливаемым. С точки зрения цены за единицу хранения это самая экономичная из всех конфигураций с избыточностью, поскольку из всего массива на обеспечение избыточности расходуется пространство, эквивалентное одному диску.

  Произвольная запись на уровне RAID 5 — затратная операция, так как реально требуется произвести две процедуры чтения записи и две операции записи для базовых дисков, чтобы рассчитать и сохранить контрольную сумму. Процедуры записи выполняются чуть быстрее, если они последовательные или количество физических дисков велико. В то же время чтение — как последовательное, так и произвольное — выполняется весьма быстро. Уровень RAID — приемлемый

выбор для томов, содержащих только данные или данные и журналы, чья рабочая нагрузка преимущественно связана с чтением, где затраты на дополнительную операцию ввода/вывода не имеют значения.

В случае отказа диска затраты производительности на уровне RAID 5 максимальны, так как информацию приходится реконструировать путем чтения всех остальных дисков. Это весьма ощутимо сказывается на производительности. Если попытаетесь сохранить доступ к серверу во время процедуры реконструкции, то не ждите ни хорошего быстродействия, ни быстрого завершения восстановления. Если вы используете RAID 5, стоит позаботиться о механизме, который при наличии проблем выполнит переключение на резервные мощности и выведет машину из эксплуатации. В любом случае целесообразно провести эталонное тестирование системы на предмет отказа диска и во время восстановления, чтобы знать, чего в этом случае ожидать. Производительность диска может ухудшиться в два раза или более при выходе из строя диска и в пять или более раз — при восстановлении, а сервер с хранилищем, который работает в 2–5 раз медленнее нормы, может повлиять на все окружающее.

К другим недостаткам можно отнести ограничения масштабируемости из-за блоков с контрольной суммой (RAID 5 начинает хуже масштабироваться уже при десяти дисках) и проблемы кэширования. Производительность RAID 5 в значительной степени зависит от кэша в RAID-контроллере, который может вступать в конфликт с потребностями сервера базы данных. Мы вернемся к этой теме чуть позже.

Один из факторов, помогающих примириться с уровнем RAID 5, — его широкая распространенность. Поэтому RAID-контроллеры часто оптимизируют именно для этого уровня, и, несмотря на теоретические ограничения, интеллектуальные контроллеры с кэшем при некоторых рабочих нагрузках работают почти так же хорошо, как контроллеры RAID 10. Правда, это может всего лишь свидетельствовать о недостаточной оптимизированности последних, но, как бы то ни было, мы с таким сталкивались.

- *RAID 10.* Уровень RAID 10 — прекрасный выбор для хранения информации. Он состоит из зеркалированных пар с данными, записанными с чередованием, так что отлично масштабируется и для чтения, и для записи. По сравнению с RAID 5 он работает и реконструируется очень быстро. К тому же этот уровень довольно легко реализуется программно.

  Падение производительности при отказе одного диска все еще остается заметным, поскольку узким местом становятся находящиеся на нем данные. В зависимости от рабочей нагрузки снижение производительности может достигать 50 %. При выборе RAID-контроллера нужно обращать внимание, не используется ли в реализации RAID 10 зеркалирование с конкатенацией. Это неоптимальное решение, так как в этом случае отсутствует чередование: может оказаться, что данные, к которым сервер обращается чаще всего, находятся только на одной паре дисков, а не распределены по многим парам, а это снижает производительность.

- *RAID 50.* Уровень RAID 50 состоит из чередующихся массивов RAID 5. Такое решение может стать неплохим компромиссом между экономичностью RAID 5 и высокой производительностью RAID 10, если имеется достаточное количество дисков. Этот метод наиболее полезен для очень больших наборов данных, например при организации хранилищ, или необычайно крупных OLTP-систем.

Различные конфигурации RAID сведены в табл. 9.2.

Таблица 9.2. Сравнение уровней RAID

Уровень	Характеристики	Избыточность	Требуется дисков	Быстрое чтение	Быстрая запись
RAID 0	Дешевый, быстрый, опасный	Нет	N	Да	Да
RAID 1	Быстрое чтение, простой, безопасный	Да	2 (обычно)	Да	Нет
RAID 5	Безопасный, быстрый, позволяющий достичь компромисса по затратам	Да	$N + 1$	Да	По-разному
RAID 10	Дорогой, быстрый, безопасный	Да	$2N$	Да	Да
RAID 50	Для очень больших хранилищ данных	Да	$2(N + 1)$	Да	Да

Отказ, восстановление и мониторинг RAID

Все уровни RAID, кроме RAID 0, обеспечивают избыточность. Это, конечно, важно, но не стоит недооценивать вероятность отказа сразу нескольких дисков. Не нужно думать, что RAID дает полную гарантию сохранности данных.

RAID не отменяет и даже не уменьшает необходимость резервного копирования. В случае возникновения проблемы время восстановления зависит от контроллера, уровня RAID, размера массива, скорости дисков и от того, требуется ли сохранять доступ к данным во время реконструкции массива.

Может случиться, что несколько дисков откажут одновременно. Например, скачок напряжения или перегрев вполне способны вывести из строя два и более диска. Но чаще бывает, что два отказа разделены небольшим промежутком времени. Нередко это остается незамеченным. Типичная ситуация — повреждение поверхности носителя в том месте, где хранятся редко используемые данные. Дефект может не проявляться много месяцев, пока кто-то не попытается прочитать данные или RAID-контроллер не воспользуется ими для реконструкции массива. Чем диск больше, тем вероятнее такое развитие событий.

Именно поэтому так важно вести мониторинг состояния RAID-массивов. Обычно вместе с контроллером поставляется программное обеспечение для формирования отчетов о состоянии массива, и им надо пользоваться, поскольку в противном случае вы так и будете пребывать в неведении об отказе диска. Возможность вовремя вос-

становить данные будет упущена, а о проблеме вы узнаете только в момент отказа второго диска, когда будет уже слишком поздно принимать меры.

Уменьшить риск можно, если регулярно активно проверять сохранность массива. Кроме того, в некоторых контроллерах реализована функция Background Patrol Read (фоновое контрольное чтение), которая ищет дефекты дисков и устраняет их, не прерывая доступа к данным. Но как и в случае восстановления, проверка очень больших массивов может занимать длительное время, поэтому планируйте ее заблаговременно.

Можно также добавить горячий резервный диск, который не используется, а сконфигурирован как резервный, чтобы контроллер мог автоматически задействовать его для восстановления. Это разумная мысль, если все серверы очень важны. Для серверов всего с 2–3 дисками такое решение дороговато, но если дисков много, то просто глупо не раскошелиться еще на один для горячей замены. Не забывайте, что вероятность отказа быстро возрастает с увеличением количества дисков.

Помимо мониторинга дисков на предмет наличия сбоев, следует также контролировать блок резервного питания аккумулятора контроллера RAID и политику кэширования записей. Если батарея выходит из строя, большинство контроллеров по умолчанию отключат кэширование записи, изменив политику кэширования на Write-Through (запись в обход кэша) вместо Write-Back (отложенная запись в кэш). Это может привести к серьезному снижению производительности. Многие контроллеры также будут периодически заряжать батареи во время процесса обучения, в течение которого кэш также отключается. Утилита управления RAID-контроллером должна позволять вам просматривать и планировать цикл зарядки батарей, чтобы он не заставал вас врасплох.

Кроме того, вы можете выполнить эталонное тестирование своей системы на предмет политики кэширования, установленной как WriteThrough. Таким образом вы узнаете, с чем столкнетесь. Возможно, придется запланировать цикл зарядки батарей на ночное время или выходные дни, перенастроить серверы, изменив переменные `innodb_flush_log_at_trx_commit` и `sync_binlog`, или просто перейти на другой сервер, позволив серверам заряжать батареи по очереди.

Выбор между аппаратной и программной реализациями RAID

Между операционной системой, файловой системой и количеством дисков, которые видит операционная система, существуют сложные зависимости. Ошибки, ограничения, да и просто неверная конфигурация могут привести к тому, что производительность будет сильно недотягивать до теоретически возможной.

Если имеется десять жестких дисков, то в идеале они могли бы параллельно обслуживать десять запросов, но иногда файловая система, операционная система или RAID-контроллер сериализуют запросы. Одно из возможных решений этой проблемы — попробовать различные конфигурации RAID. Например, если вы хотите

воспользоваться зеркалированием для обеспечения избыточности и высокой производительности, то можно сконфигурировать десять дисков следующими способами.

- Сконфигурировать один том уровня RAID 10, состоящий из пяти зеркалированных пар. Операционная система будет видеть единый большой том, а RAID-контроллер скроет десять составляющих его дисков.
- Сконфигурировать пять зеркалированных пар уровня RAID 1, так что операционная система будет адресовать пять томов вместо одного.
- Сконфигурировать пять зеркалированных пар уровня RAID 1 в RAID-контроллере, а затем использовать программную реализацию RAID 0, чтобы представить все пять томов в виде одного логического тома. В результате получится массив RAID 10, реализованный отчасти программно, а отчасти аппаратно.

Какой вариант лучше? Зависит от взаимодействия между всеми компонентами системы. Возможно, эти конфигурации будут работать с одинаковой производительностью, а может, и нет.

Мы наблюдали сериализацию в различных конфигурациях. Например, в файловой системе ext3 имеется по одному мьютексу для каждого индексного дескриптора, поэтому конфигурирование InnoDB с параметром `innodb_flush_method=O_DIRECT` (обычная конфигурация) приведет к блокировкам на уровне индексных узлов в файловой системе. Тем самым конкурентный ввод/вывод в файлы станет невозможным, а система будет работать хуже, чем способна теоретически.

Аналогичная ситуация наблюдалась, когда мы работали с томом RAID 10 из десяти дисков, файловой системой ReiserFS и подсистемой хранения InnoDB с включенным параметром `innodb_file_per_table`, но теперь сериализации подвергались запросы к каждому устройству. После перехода на уровень RAID 0 поверх аппаратного RAID 1 пропускная способность возросла в пять раз, поскольку подсистема хранения стала вести себя так, будто вместо одного диска появилось пять. Это явление было вызвано ошибкой, впоследствии исправленной, но сам факт хорошо иллюстрирует, что вообще может произойти.

Сериализация может наблюдаться на любом уровне программного или аппаратного стека. Столкнувшись с такой проблемой, можно попытаться изменить файловую систему, обновить ядро, предъявить операционной системе больше устройств или организовать другое сочетание программной и аппаратной реализации RAID. Следует также посмотреть, что утилита iostat говорит о конкурентном доступе к устройству, и убедиться в том, что ввод/вывод действительно производится конкурентно (подробнее этот вопрос будет рассмотрен в дальнейшем).

И не забывайте при настройке нового сервера об эталонном тестировании! Оно позволит доказать, что фактическая производительность совпадает с ожидаемой. Например, если один диск может выполнять 200 операций с произвольным доступом в секунду, то том RAID 10 из восьми дисков должен выполнять примерно 1600 таких операций. Если наблюдается существенно меньшая величина, например всего 500 операций чтения, то проблему необходимо исследовать. Следите за тем, чтобы эталонные тесты нагружали подсистему ввода/вывода так же, как это делает

MySQL, — например, установите флаг `O_DIRECT` и тестируйте производительность ввода/вывода на одном файле, если используется InnoDB с отключенным режимом `innodb_file_per_table`. Обычно для проверки правильности настройки нового оборудования мы используем инструмент sysbench.

Конфигурация RAID и кэширование

Обычно для конфигурирования контроллера RAID нужно войти в его программу настройки на стадии начальной загрузки компьютера. Большинство контроллеров предлагают множество разнообразных параметров, но мы остановимся только на размере фрагмента для массивов с чередованием и *кэше контроллера* (его еще называют *RAID-кэшем*, будем считать эти термины синонимами).

Размер фрагмента для слоя RAID

Оптимальный размер фрагмента для чередования зависит от рабочей нагрузки и оборудования. Теоретически для ввода/вывода с произвольной выборкой лучше подходит большой размер фрагмента, поскольку это означает, что один диск способен удовлетворить более длинные запросы на чтение.

Чтобы понять, почему это так, рассмотрим типичную операцию произвольного ввода/вывода для имеющейся рабочей нагрузки. Если размер фрагмента не меньше длины этой операции и данные не пересекают границу между фрагментами, то в чтении может участвовать только один диск. Но если размер фрагмента меньше длины считываемых данных, то придется задействовать несколько дисков.

Но оставим теорию. На практике многие RAID-контроллеры плохо работают с большими фрагментами. Например, контроллер может использовать фрагмент как единицу кэширования в своем кэше, а это бывает расточительно. Он может также сопоставить размер фрагмента, размер кэша и размер единицы считывания (объем данных, считываемых за одну операцию). Если единица считывания оказывается слишком велика, то кэш используется неэффективно, так как в него попадает гораздо больше данных, чем необходимо, даже для крохотных запросов.

Кроме того, на практике сложно узнать, расположен ли некоторый элемент данных на одном или нескольких дисках. Даже при размере фрагмента 16 Кбайт, что соответствует размеру страницы InnoDB, мы не можем быть уверены в том, что все операции считывания выровнены по границе 16 Кбайт. Файловая система может фрагментировать файл и обычно выравнивает его фрагменты по границам блока файловой системы, который чаще всего составляет 4 Кбайт. Некоторые файловые системы ведут себя разумнее, но рассчитывать на это не стоит.

Вы можете настроить систему так, чтобы блоки были полностью выровнены, начиная с приложения и заканчивая основным хранилищем данных: блоки InnoDB, блоки файловой системы, LVM, смещение раздела, секция RAID и секторы диска. Эталонные тесты показали, что если все блоки выровнены, это может улучшить производительность приблизительно на 15 и 23 % для произвольного чтения и записей соответственно.

Точные методики для выравнивания всех блоков слишком специфичны, поэтому не станем описывать их здесь, однако об этом можно найти много полезной информации, например, в нашем блоге по адресу http://www.mysqlperformanceblog.com.

RAID-кэш

RAID-кэш представляет собой небольшую (относительно) область памяти, которая физически находится на плате RAID-контроллера. Его можно использовать для буферизации данных на пути между диском и хост-системой. Далее перечислены несколько причин, по которым RAID-контроллер может воспользоваться кэшем.

- *Кэширование результатов чтения.* Прочитав какие-то данные с дисков и отправив их хост-системе, контроллер может эти данные сохранить, тогда последующие запросы тех же самых данных можно будет удовлетворить, не обращаясь снова к диску.

  Как правило, это неудачное использование RAID-кэша. Почему? Потому что у операционной системы и сервера базы данных есть свои, куда более обширные кэши. Если произойдет попадание в один из них, то до RAID-кэша дело даже не дойдет. И наоборот, если не было попадания ни в один из вышележащих кэшей, вероятность того, что данные обнаружатся в RAID-кэше, окажется исчезающе мала. Поскольку RAID-кэш намного меньше, то почти наверняка нужные данные уже вытеснены из него и заменены новыми. В общем, как ни крути, сохранять результаты чтения в RAID-кэше — пустая трата памяти.
- *Упреждающее кэширование данных при чтении.* Если RAID-контроллер обнаруживает запросы на чтение последовательных данных, то он может прибегнуть к упреждающему чтению, то есть заранее выбрать данные, которые, вероятнее всего, скоро понадобятся. Однако до тех пор, пока запрос не поступил, эти данные нужно где-то хранить. Для этой цели вполне подходит RAID-кэш. Влияние такой тактики на производительность может варьироваться в широких пределах, поэтому следует проверять, дала ли она что-нибудь в вашей ситуации. Упреждающее чтение на уровне RAID-контроллера может не иметь никакого эффекта, если сервер базы данных реализует собственный алгоритм интеллектуального упреждающего чтения (как, например, InnoDB). К тому же оно может воспрепятствовать гораздо более важному механизму буферизации синхронных операций записи.
- *Кэширование операций записи.* RAID-контроллер может буферизовать в своем кэше операции записи, откладывая их выполнение на более позднее время. У такой методики есть два достоинства: во-первых, контроллер может вернуть хост-системе признак успешного завершения быстрее, чем если бы физически производил запись на диски, а во-вторых, операции записи можно объединить и выполнить более эффективно.
- *Внутренние операции.* Некоторые операции RAID-контроллера очень сложны, особенно запись в случае RAID 5, когда нужно вычислять контрольную сумму, которая позволит реконструировать данные в случае отказа диска. Для выполнения таких операций контроллеру необходима память.

В частности, по этой причине в некоторых контроллерах уровень RAID 5 работает медленно: для обеспечения высокой производительности контроллер должен прочитать в кэш много данных, но не все контроллеры умеют разумно распределять память кэша между операциями записи и операциями вычисления контрольной суммы для RAID 5.

Вообще говоря, память на плате RAID-контроллера — это дефицитный ресурс, которым нужно распорядиться с толком. Использовать его для кэширования результатов чтения — обычно откровенное расточительство, а вот использование его для кэширования операций записи может ощутимо повысить производительность ввода/вывода. Многие контроллеры позволяют указать, как распределить эту память. Например, можно выбрать, какую часть отвести под кэширование операций записи, а какую — для результатов чтения. В случае RAID 0, RAID 1 и RAID 10 лучше всего выделить все 100 % памяти контроллера под кэширование операций записи. В случае же RAID 5 следует зарезервировать часть памяти контроллера для внутренних операций. В общем случае это хорошая рекомендация, но применима она не всегда: разные RAID-контроллеры конфигурируются по-разному.

Если RAID-кэш применяется для кэширования операций записи, то многие контроллеры позволяют указать, на какое время можно задерживать запись (1 секунда, 5 секунд и т. д.). Чем больше задержка, тем больше операций записи удастся сгруппировать и сбросить на диск оптимальным образом. Недостаток же заключается в том, что запись оказывается более «пульсирующей». В этом нет ничего страшного, если только приложение не отправит пачку запросов на запись как раз в тот момент, когда кэш контроллера заполнен и должен быть сброшен на диск. Если для запросов приложения не осталось места, ему придется ждать. Если уменьшить задержку, то физических операций записи будет больше и их группировка окажется менее эффективной, зато пики удастся сгладить и кэш сумеет справиться с внезапным всплеском количества запросов от приложения. (Мы несколько упростили изложение — в контроллерах часто реализуются сложные патентованные алгоритмы балансировки, но мы пытаемся объяснить основополагающие принципы.)

Кэш записи очень полезен для синхронных операций, например для системных вызовов `fsync()` при записи в журнал транзакций и создании двоичного журнала в режиме `sync_binlog`, но его не следует активизировать, если контроллер не оборудован блоком аварийного электропитания (battery backup unit, BBU) или другим энергонезависимым запоминающим устройством[1]. В противном случае внезапное отключение напряжения может легко привести к повреждению базы данных и даже транзакционной файловой системы. Однако при наличии BBU включение кэша записи может повысить производительность в 20 и более раз, если рабочая нагрузка подразумевает большое количество сбросов в журнал, например, в момент фиксации транзакций.

И в завершение следует отметить, что многие накопители на жестких дисках оборудованы собственным кэшем записи, который может «обжуливать» операцию `fsync()`,

[1] Существует несколько технологий, в том числе конденсаторы и флеш-накопители, но здесь мы сосредоточимся на BBU.

обманывая контроллер сообщением, будто данные записаны на физический носитель. Жесткие диски, подключенные напрямую, а не через RAID-контроллер, позволяют операционной системе управлять своими кэшами, но это работает не всегда. Такие кэши обычно сбрасываются при вызове `fsync()` и обходятся при синхронном вводе/выводе, но все-таки накопитель может лгать. Следует либо убедиться в том, что кэш действительно сбрасывается в момент вызова `fsync()`, либо отключить его вовсе, поскольку аварийное питание от батареи для него не предусмотрено. Накопители, которыми операционная система или микропрограмма RAID-контроллера не может корректно управлять, не раз становились причиной потери данных.

По этой и другим причинам всегда целесообразно, устанавливая новое оборудование, провести настоящий краш-тест (в буквальном смысле — выдернув шнур из розетки). Зачастую это единственный способ найти трудноуловимые ошибки в конфигурации или воспроизвести коварное поведение накопителя. Удобный скрипт для этих целей можно найти по адресу http://brad.livejournal.com/2116715.html.

Если вы хотите быть абсолютно уверенными в надежности BBU, которым оснащен RAID-контроллер, оставьте шнур выдернутым на длительное время. Некоторые батарейные источники поддерживают питание не так долго, как заявлено в спецификации. В этом случае, как и во многих других, одно слабое звено может привести к бесполезности всей цепочки компонентов хранения данных.

Сети хранения данных и сетевые системы хранения данных

Сети хранения данных (storage area networks, SAN) и *сетевые системы хранения данных* (network-attached storage, NAS) — это два взаимосвязанных способа подключения внешних устройств хранения файлов к серверу. Различие состоит в способе доступа к хранилищу. SAN предоставляет такой интерфейс на уровне блоков, что серверу кажется, будто устройство подключено напрямую. Что же касается NAS-устройств, то они работают по протоколу файлового уровня, например, NFS или SMB. SAN обычно подключается к серверу по протоколу Fibre Channel Protocol (FCP) или iSCSI, тогда как NAS-устройства — по стандартному сетевому соединению. Некоторые системы хранения, такие как, например, NetApp Filer, могут использовать оба способа.

В дальнейшем аббревиатурой SAN мы будем обозначать оба этих типа хранилищ, не забывайте об этом во время чтения. Основное различие между ними заключается в доступе к хранилищу в виде файлов или блоков.

SAN-сети разрешают серверу получать доступ к очень большому количеству жестких дисков — как правило, 50 и более — и обычно имеют большие интеллектуальные кэши для буферизации записей. Интерфейс уровня блока отображается на сервере как номера логических единиц (LUN) или виртуальные тома (если вы не используете NFS). Многие SAN-сети также позволяют группировать несколько узлов в кластеры для повышения производительности или увеличения емкости хранилища.

Нынешнее поколение SAN-сетей отличается от тех, которые были доступны несколько лет назад. У многих новых SAN-сетей имеется гибридное хранилище данных, состоящее из флеш-памяти и жесткого диска, а не только жесткие диски. Часто у них есть флеш-кэши размером 1 Тбайт или больше, в отличие от более старых SAN-сетей, у которых были относительно небольшие кэши. Кроме того, старые SAN-сети не могли не увеличить буферный пул с помощью большого кэша, тогда как новые могут не увеличивать. Таким образом, в некоторых ситуациях новые SAN-сети могут обеспечить лучшую производительность, чем старые.

Эталонные тесты SAN-сетей

Мы провели эталонное тестирование разнообразных продуктов, предлагаемых различными производителями. В табл. 9.3 показана выборка типичных результатов в условиях низкой конкуренции.

Таблица 9.3. Количество синхронных однопоточных операций в секунду с данными размером 16 Кбайт для файла размером 4 Гбайт

Устройство	Последовательная запись	Последовательное чтение	Произвольная запись	Произвольное чтение
SAN1 с RAID 5	2428	5794	629	258
SAN1 с RAID 10	1765	3427	1725	213
SAN2 поверх NFS	1768	3154	2056	166
10k RPM жесткие диски, RAID 1	7027	4773	2302	310
Intel SSD	3045	6266	2427	4397

Конкретные производители и конфигурации SAN-сетей останутся в секрете, хотя можем сказать, что это не низкобюджетные SAN-сети. Мы провели эти эталонные тесты с синхронными операциями с данными размером 16 Кбайт, тем самым эмулируя то, как InnoDB работает при настройке в режиме `O_DIRECT`.

Какие выводы можно сделать на основе анализа табл. 9.3? Не все протестированные системы можно напрямую сопоставить, поэтому не стоит внимательно изучать конкретные значения. Однако результаты хорошо иллюстрируют общую производительность, которой можно было ожидать от этих типов устройств. SAN-сети могут поглощать множество последовательных записей, поскольку способны буферизовать и объединять их. Они могут без проблем обслуживать последовательные чтения, поскольку умеют их предсказывать, осуществлять предварительную выборку и обслуживать конкретные чтения уже из кэша. При произвольной записи наблюдается некоторое замедление, поскольку записи нельзя объединить таким же образом. При произвольном чтении SAN-сети продемонстрировали крайне низкие результаты: такое чтение обычно сопровождается непопаданием в кэш, поэтому приходится ждать ответа от жестких дисков. Кроме того, наблюдается задержка транспортировки между сервером и SAN. Именно поэтому SAN-сеть, подключенная

через NFS, выполняет меньше произвольных чтений в секунду, чем один локально подключенный жесткий диск.

Мы провели эталонное тестирование с файлами больших размеров, однако у нас нет результатов по этим файлам для всех вышеперечисленных систем. Результат, однако, всегда предсказуем: независимо от того, насколько большие и мощные SAN-сети используются, получить небольшое время отклика или хорошую пропускную способность для небольших произвольных операций не удается. Дело в том, что происходит слишком много задержек из-за существования расстояния между сервером и SAN-сетями.

Наши эталонные тесты показали пропускную способность в операциях в секунду, однако в них не учитываются все факторы. Есть как минимум три других важных показателя: пропускная способность в байтах в секунду, конкурентный доступ и время отклика. В целом по сравнению с напрямую подключенным хранилищем SAN-сети обеспечат хорошую последовательную пропускную способность в байтах в секунду как для чтения, так и для записи. Большинство SAN-сетей могут поддерживать высокую конкуренцию, однако мы провели эталонное тестирование только одного потока, чтобы показать самый плохой вариант. Но когда рабочее множество не помещается в кэши SAN-сетей, произвольное чтение будет очень плохим с точки зрения пропускной способности и задержек. Но даже если рабочее множество поместить в кэш, задержки будут выше, чем при прямом подключении.

Использование SAN-сетей через NFS или SMB

Некоторые SAN, такие как программа работы с файлами NetApp, обычно получают доступ к NFS вместо Fibre Channel или iSCSI. Раньше этого старались избежать, но сейчас NFS работает намного лучше. Вы можете добиться приличной производительности при NFS, хотя для этого нужно специально настроить сеть. Производители SAN-сетей рекомендуют приемы, которые должны помочь вам в настройке.

Основное соображение заключается в том, как сам протокол NFS влияет на производительность. Многие операции с файлами метаданных, которые обычно выполняются в памяти в локальной файловой системе или в SAN без NFS, могут требовать запроса и ответа по сети с NFS. Например, мы отметили серьезное снижение производительности при хранении двоичных журналов в NFS, даже с отключенным протоколом `sync_bin log`. Это связано с тем, что добавление к двоичному журналу увеличивает его размер, что требует операции метаданных, которая вызывает дополнительные запросы и ответы.

Доступ к SAN или NAS можно также получить по протоколу SMB. В этом случае справедливы аналогичные соображения: может потребоваться множество чувствительных к задержкам запросов и ответов. Это не имеет большого значения для обычного пользователя настольного компьютера, который хранит электронные таблицы или другие документы на подключенном к нему накопителе, или даже для таких операций, как перенесение резервных копий на другой сервер, но это может быть серьезной проблемой для способа, которым MySQL читает и записывает файлы.

Производительность MySQL в SAN

Эталонные тесты ввода/вывода — это один из способов взглянуть на ситуацию, но как насчет производительности MySQL в SAN? Во многих случаях MySQL работает очень хорошо и вы не столкнетесь с ситуациями, когда использование SAN-сетей приведет к некоторому ухудшению производительности. Тщательное логическое и физическое проектирование, включая индексирование и соответствующее серверное оборудование (много памяти!), позволит избежать большого числа произвольных операций ввода/вывода или преобразовать их в последовательные. Тем не менее вы должны знать, что такая система может поддерживать неустойчивое равновесие в течение некоторого периода времени, которое легко может быть нарушено новым запросом, сменой схемы или редко выполняемой операцией.

Например, один наш знакомый, пользователь SAN-сетей был вполне доволен обычной производительностью, пока не захотел удалить много строк из старой таблицы, которая очень сильно выросла. Команда `DELETE`, удалявшая лишь пару сотен строк в секунду, работала очень долго, поскольку каждая строка требовала произвольного ввода/вывода, который SAN-сети не могли выполнить быстро. Не было никакого способа ускорить операцию, пришлось долго ждать ее завершения. Другим сюрпризом для того же пользователя стала ситуация, когда команда `ALTER` также очень сильно замедлилась для большой таблицы.

Это типичные примеры того, что не работает на SAN-сетях как полагается: однопоточные задачи, выполняющие множество произвольных операций ввода/вывода. Репликация — еще одна однопоточная задача для текущих версий MySQL, и велика вероятность того, что в результате ее выполнения подчиненные серверы, данные которых хранятся в SAN, могут отстать от главного. Пакетные задания также могут начать работать медленнее. Возможно, вы сможете выполнять одиночные операции, чувствительные к задержке, в часы с низкой нагрузкой или в выходные дни, но постоянные элементы сервера, такие как репликация, двоичные журналы и журналы транзакций InnoDB, постоянно требуют хорошей производительности при небольших и/или произвольных операциях ввода/вывода.

Надо ли использовать SAN

Ах, это вечный вопрос, а в некоторых случаях — вопрос на миллион долларов. Есть много факторов, которые следует учитывать, и некоторые из них мы перечислим.

- *Резервные копии*. Централизованное хранилище может упростить управление резервными копиями. Когда все хранится в одном месте, вы можете просто выполнить резервное копирование SAN, будучи при этом уверенными, что все данные скопированы. Тем самым такие вопросы, как «Вы уверены, что мы создали резервную копию всех наших данных?», становятся тривиальными. Кроме того, в некоторых устройствах реализованы возможность непрерывной защиты данных (continuous data protection, CDP) и инструменты выполнения моментальных снимков, которые делают создание резервных копий намного проще и гибче.

- *Упрощенное планирование мощности.* Не знаете, сколько мощностей вам требуется? SAN-сети дают возможность купить хранилище оптом, разделить его, изменять его размер и распространять по требованию.
- *Что лучше, консолидация хранилищ или консолидация серверов.* Некоторые IT-директора анализируют работу своих центров обработки данных и делают вывод о том, что большая часть ресурсов ввода/вывода простаивает, причем с точки зрения как объема памяти, так и операций ввода/вывода. Не будем приводить здесь аргументов, но если вы для улучшения использования хранилища его централизуете, то как это повлияет на системы, использующие хранилище? Производительность типичных операций с базой данных может различаться буквально на порядок, и в результате вы можете обнаружить, что необходимо запустить в десять раз больше серверов (а то и больше) для обработки рабочей нагрузки. И хотя пропускная способность ввода/вывода центра обработки данных может использоваться в SAN намного лучше, это может быть достигнуто в ущерб использованию многих других систем (сервер базы данных тратит много времени на ожидание ввода/вывода, сервер приложений тратит много времени на ожидание базы данных и т. д.). На практике мы обнаруживали множество возможностей для консолидации серверов и сокращения затрат путем *децентрализации* хранилища.
- *Высокая надежность.* Иногда люди рассматривают SAN-сети как высоконадежное решение. В главе 12 мы выскажем предположение, что это связано с непониманием того, что означает высокая надежность.

  Наш опыт свидетельствует о том, что в SAN-сетях довольно часто наблюдаются сбои и простои. Это происходит не потому, что они ненадежны (они вполне надежны), а потому, что люди не хотят верить, что такое чудо техники может сбоить. Кроме того, SAN-сети иногда представляют собой загадочный сложный черный ящик, и если что-то идет не так и в них возникнет проблема, никто не знает, как ее устранить. Получение опыта, позволяющего управлять сетями SAN, может быть непростым и недешевым удовольствием. Большинство SAN-сетей непрозрачны, поэтому вы не должны слепо доверять их администратору, персоналу службы поддержки или консоли управления. Мы видели случаи, когда все трое ошибались и у SAN возникала проблема — неисправный жесткий диск, что вело к ухудшению производительности[1]. Это еще одна причина для знакомства с утилитой sysbench: вы можете набросать эталонный тест ввода/вывода, чтобы доказать или опровергнуть вину сетей SAN.
- *Взаимодействие между серверами.* Разделение хранилища может привести к тому, что независимые на первый взгляд системы будут влиять друг на друга, иногда очень сильно. Например, один знакомый пользователь сетей SAN был потрясен, когда интенсивная операция ввода/вывода на сервере разработки привела практически к остановке сервера базы данных. Пакетные задания, `ALTER TABLE`, резервные копии — все, что требует множества операций ввода/вывода в одной системе, может вызвать зависание в других системах. Иногда негативное воздействие намного

[1] Веб-консоль управления SAN настаивала на том, что все жесткие диски нормальные, пока мы не попросили администратора нажать Shift+F5 для отключения кэша браузера принудительной перезагрузки консоли!

хуже, чем вы могли предположить: кажущаяся незначительной рабочая нагрузка неожиданно может вызвать серьезное падение производительности.

- *Стоимость*. Стоимость чего? Затраты на управление и администрирование? Затраты на операцию ввода/вывода в секунду (IOPS)? Цена наклейки?

  Есть веские причины использовать SAN, но независимо от того, что говорят продавцы, высокая производительность — по крайней мере производительность того типа, который требуется MySQL, — не является веской причиной. (Выберите поставщика сетей SAN, позвоните ему — и вам, скорее всего, скажут, что в целом они согласны с этим тезисом, но затем добавят, что их продукт является исключением из правила.) Если вы будете рассматривать производительность и цену вместе, картина станет еще наглядней (если вы хотите получить хорошее соотношение цены и производительности, знайте: флеш-память или старомодные жесткие диски с хорошим RAID-контроллером и кэш-памятью с резервной батареей обеспечивают гораздо лучшую производительность по гораздо более низкой цене).

  И не забудьте попросить продавца назвать цену двух SAN. Вам нужны как минимум две, поскольку в противном случае вы приобретете лишь дорогой источник сбоев.

Мы могли бы рассказать много страшных историй, однако не ставим перед собой цель отпугнуть вас от использования SAN-сетей. Большинство пользователей сетей SAN, которых мы знаем, без ума от них! Когда вы пытаетесь ответить на вопрос, стоит ли вам использовать SAN-сети, самое главное — четко понять, какие проблемы вы хотите решить. SAN-сети могут делать многое, но, как правило, решение проблемы с производительностью в этот список не входит. Напротив, SAN-сети могут быть отличным решением, если вам не требуется большое количество высокопроизводительных произвольных операций ввода/вывода, но интересуют такие функции, как моментальные снимки, консолидация хранилищ, устранение избыточности данных и виртуализация.

В результате большинство веб-приложений не используют сети SAN для баз данных, но они очень популярны для так называемых корпоративных приложений. Обычно корпорации располагают большим бюджетом, поэтому могут позволить себе предметы роскоши, такие как SAN. (Иногда SAN-сети воспринимаются как символ статуса!)

Использование нескольких дисковых томов

Рано или поздно возникает вопрос о том, где размещать данные. MySQL создает различные файлы:

- файлы данных и индексов;
- журналы транзакций;
- двоичные журналы;
- журналы общего назначения (журналы ошибок, запросов, медленных запросов);
- временные файлы и таблицы.

В MySQL встроено не так уж много возможностей для управления табличным пространством. По умолчанию все файлы, принадлежащие одной базе данных (схеме), помещаются в один каталог. Есть несколько возможностей более точно указать местоположение данных. Так, можно задать, куда поместить индекс над таблицей типа MyISAM, а в версии MySQL 5.1 к вашим услугам секционированные таблицы.

В подразумеваемой по умолчанию конфигурации InnoDB все данные и индексы размещаются в одном наборе файлов, а в каталоге базы данных хранятся лишь файлы с определениями таблиц. Поэтому обычно все данные и индексы помещают в один том.

Однако в некоторых случаях, для того чтобы справиться с высокой нагрузкой ввода/вывода, имеет смысл задействовать несколько томов. Например, если имеется пакетное задание, которое записывает данные в объемную таблицу, то лучше разместить последнюю в отдельном томе, чтобы не отнимать у других запросов драгоценную пропускную способность подсистемы ввода/вывода. В идеале следует проанализировать доступ операций ввода/вывода к разным частям данных и разместить их соответствующим образом, но сделать это проблематично, если только данные уже не находятся в разных томах.

Возможно, вы слышали стандартный совет: помещать журналы транзакций и файлы данных в разные тома, чтобы последовательная запись в журналы не мешала произвольному вводу/выводу. Однако, если у вас не слишком много жестких дисков (около 20) или флеш-память, стоит хорошенько подумать, прежде чем следовать этому совету.

Реальная выгода от разделения журналов и файлов данных состоит в том, что уменьшается вероятность потерять одновременно и то и другое в случае отказа. Помещение их на разные диски — это хороший прием, если RAID-контроллер не оснащен кэшем записи с аварийным питанием. Но если такой кэш есть, то выделение отдельного тома бывает оправданно не так часто, как может показаться. Высокая производительность редко является определяющим фактором. Объясняется это тем, что, хотя запись в журналы транзакций производится часто, большинство операций очень короткие. В результате кэш RAID обычно объединяет запросы, и получается лишь несколько последовательных запросов на физические операции записи в секунду. Этому вряд ли может помешать ввод/вывод в файлы данных с произвольным доступом, если вы не насыщаете полностью контроллер RAID. Для журналов общего назначения характерны асинхронная последовательная запись и низкая интенсивность, так что они тоже вполне могут разделять один том с данными.

На эту проблему можно взглянуть и под другим углом зрения, которым часто пренебрегают. Улучшается ли производительность в результате размещения журналов в отдельных томах? Как правило, да, но стоит ли оно того? Ответ часто отрицательный, и вот почему: выделять специальные диски для журналов транзакций затратно. Предположим, что всего дисков шесть. Напрашивающиеся решения: объединить все шесть в один RAID-том или отдать четыре под данные, а два — под журналы транзакций. Но если выбрать второй вариант, то количество дисков для хранения данных уменьшится на треть, а это немало, притом что два диска выделены под тривиальную

рабочую нагрузку (в предположении, что RAID-контроллер оборудован кэшем записи с резервным питанием).

Но если жестких дисков много, то относительная стоимость выделения части из них под журналы транзакций уменьшается, и такое решение может стать оправданным. Например, если имеется 30 дисков, то, отведя под журналы два из них, сконфигурированных как том уровня RAID 1, можно будет обеспечить максимально быструю запись. Можно даже еще повысить производительность, зарезервировав часть RAID-кэша под запись только для этого тома.

Экономичность не единственное соображение, которое стоит учитывать. Еще одна причина хранить данные и журналы транзакций InnoDB в одном томе состоит в том, что при такой стратегии вы сможете использовать функцию мгновенного снимка подсистемы управления логическими томами (LVM) для резервного копирования без блокировки. Некоторые файловые системы допускают снятие согласованных мгновенных снимков с нескольких томов, и для таких систем данное преимущество не очень существенно, но в случае системы ext3 его следует иметь в виду. (Для резервного копирования без блокировки можно также использовать Percona XtraBackup — подробности см. в главе 15.)

В режиме `sync_binlog` двоичные журналы с точки зрения производительности аналогичны журналам транзакций. Однако хранить двоичные журналы отдельно от данных — действительно хорошая идея: это безопаснее, поскольку в таком случае они могут уцелеть даже после потери данных. Следовательно, ничто не помешает использовать их для восстановления состояния на определенный момент в прошлом. Это соображение неприменимо к журналам транзакций InnoDB, так как последние без файлов данных бесполезны — невозможно применить журналы транзакций к резервной копии, снятой прошлой ночью. (Это различие между журналами транзакций и двоичными журналами может показаться искусственным администратору, привыкшему к другим базам данных, где они представляют собой одно и то же.)

Есть еще один распространенный случай, когда имеет смысл выделять для некоторых файлов отдельное место. Речь идет о специальном каталоге, который MySQL использует для файловых сортировок и создания временных таблиц на диске. Если генерируемые при этом файлы не слишком велики, то, возможно, стоит размещать их во временной файловой системе в памяти, например такой, как tmpfs. Это самый быстрый вариант. Если он вам не подходит, то создавайте временный каталог на том же устройстве, на котором находится операционная система.

Типичное распределение дискового пространства таково: операционная система, раздел свопинга и двоичные журналы — в томе уровня RAID 1, а для всего остального — том уровня RAID 5 или RAID 10.

Конфигурация сети

Так же как производительность жесткого диска ограничена временем задержки и пропускной способностью, для сетевого соединения лимитирующими факторами являются задержка и полоса пропускания (что по сути то же самое, что и пропускная способность). Для большинства приложений основную проблему составляет время

задержки: типичное приложение выполняет много коротких операций передачи по сети, поэтому незначительные задержки для каждой транзакции суммируются.

Серьезным узким местом может стать плохо работающая сеть. Потеря пакетов — это обычная проблема. Даже 1 % потерянных пакетов существенно снижает производительность, так как различные уровни стека протоколов будут пытаться исправить ошибку, для чего ненадолго перейдут в режим ожидания, а затем начнут отправлять пакет повторно, затрачивая на все это лишнее время. Еще одна типичная проблема — неправильно сконфигурированный или медленно работающий DNS-сервер.

DNS — это ахиллесова пята, поэтому на промышленных серверах целесообразно включать режим `skip_name_resolve`. Неработающий или медленно работающий DNS-сервер — беда для многих приложений, но для MySQL особенно. Получая запрос на соединение, MySQL выполняет прямой и обратный поиск в DNS. Он может завершиться неудачно по самым разным причинам, и в таком случае в установлении соединения будет вообще отказано или данная процедура займет недопустимо много времени, что в общем случае может привести к полному хаосу, в том числе стать способом DoS-атаки. Если включить параметр `skip_name_resolve`, то MySQL не будет посылать никаких запросов DNS-серверу. Но это означает, что во всех учетных записях пользователей столбец `host` может содержать только IP-адрес (возможно, с метасимволами) или строку localhost. Пользователь, в учетной записи которого указано доменное имя, не сможет соединиться с сервером.

Другим распространенным источником проблем в типичных веб-приложениях является запаздывание TCP, которое можно настроить с помощью параметра `back_log MySQL`. Этот параметр управляет размером очереди MySQL для входящих TCP-соединений. В средах, где каждую секунду создается и уничтожается множество соединений, установленного значения по умолчанию, равного 50, недостаточно. Симптомом является то, что клиент время от времени видит ошибку «Соединение отклонено» в сочетании с трехсекундными тайм-аутами. Для систем с большим объемом работы этот параметр следует увеличить. Кажется, что не будет вреда в увеличении его до сотен или даже тысяч, однако в таком случае вам, скорее всего, придется изменить также настройки сети TCP вашей операционной системы. В системах GNU/Linux необходимо увеличить лимит параметра `somaxconn` (по умолчанию установлено значение `128`) и проверить параметры `log tcp_max_syn_back` в sysctl (чуть позже приведем соответствующий пример).

Чтобы добиться хорошей производительности, необходимо специально настраивать сеть, а не соглашаться с параметрами по умолчанию. Для начала проанализируйте количество переходов между узлами и начертите карту физического расположения сети. Пусть, например, имеется десять веб-серверов, подключенных к коммутатору Web посредством гигабитной сети Ethernet (1 GigE), и этот коммутатор соединен с коммутатором Database такой же гигабитной сетью. Не потратив время на трассировку соединений, вы так никогда и не узнаете, что полная пропускная способность сети, соединяющей веб-серверы с серверами баз данных, ограничена 1 Гбит! Кроме того, каждый переход увеличивает задержку.

Целесообразно вести мониторинг производительности и ошибок сети на всех портах. Это относится ко всем портам на серверах, маршрутизаторах и коммутаторах. Программа Multi Router Traffic Grapher (MRTG, http://oss.oetiker.ch/mrtg/) — проверенное на практике решение с открытым кодом для мониторинга устройств. Упомянем также две утилиты для мониторинга производительности сети (а не самих устройств): Smokeping (http://oss.oetiker.ch/smokeping/) и Cacti (http://www.cacti.net).

При организации сетей большое значение имеет физическая удаленность. В междугородных сетях задержка намного больше, чем в локальной сети центра обработки данных, даже если технически пропускная способность одинакова. Если узлы находятся очень далеко друг от друга, то приходится учитывать и скорость распространения света. Например, если центры обработки данных находятся на Западном и Восточном побережье США, они отстоят друг от друга примерно на 4800 км. Поскольку скорость света составляет 300 000 километров в секунду, то прохождение пакета в одну сторону займет никак не меньше 16 миллисекунд, а в обе стороны — по крайней мере 32 миллисекунды. Но на производительность влияет не только физическое расстояние — между крайними точками находятся и другие устройства. Повторители, маршрутизаторы, коммутаторы — все они в какой-то мере снижают производительность. К тому же чем больше расстояние между узлами сети, тем менее предсказуемо и надежно поведение соединяющих их каналов связи.

Стоит всеми силами избегать операций, требующих передачи данных между центрами обработки в реальном времени[1]. Если это невозможно, проектируйте приложение так, чтобы оно адекватно реагировало на сетевые ошибки. Например, следует ограничить количество процессов, порождаемых веб-сервером Apache, поскольку они могут долго ждать соединения с удаленным центром обработки данных по каналу связи с большим процентом потери пакетов.

В локальных сетях применяйте хотя бы технологию 1 GigE. Для прокладки магистрального канала между коммутаторами может потребоваться и сеть типа 10 GigE. Если нужна еще бóльшая пропускная способность, то можно воспользоваться *сетевым транкингом* — соединить несколько сетевых карт (NIS) для повышения величины канала. Транкинг — это, по сути дела, распараллеливание потоков данных в сети, он также может оказаться очень полезным как часть стратегии обеспечения высокой доступности.

Если необходима чрезвычайно высокая пропускная способность, то, возможно, удастся улучшить производительность путем настройки сетевых параметров операционной системы. Если соединений немного, но запросы или результирующие наборы велики, то можно увеличить размер буфера TCP. В разных системах это делается по-разному, но в большинстве систем GNU/Linux следует изменить значения параметров в файле `/etc/sysctl.conf` и выполнить команду `sysctl-p` либо воспользоваться

[1] Репликация не считается передачей данных между центрами обработки в реальном времени. Поэтому вполне здравой выглядит мысль реплицировать данные в удаленный центр обработки ради повышения безопасности. Подробнее эта тема будет рассмотрена в главе 10.

файловой системой /proc, записав новые значения в файлы, находящиеся в каталоге /proc/sys/net/ с помощью команды echo. Хорошие пособия по этой теме можно найти в Сети по запросу TCP tuning guide («Тонкая настройка TCP»).

Но чаще возникает необходимость отрегулировать настройки для эффективной работы с большим количеством соединений и маленькими запросами. Очень распространена настройка диапазона локальных портов. Вот как система конфигурируется по умолчанию:

```
[root@server ~]# cat /proc/sys/net/ipv4/ip_local_port_range
32768   61000
```

Иногда диапазон портов нужно расширить, например:

```
[root@caw2 ~]# echo 1024 65535 > /proc/sys/net/ipv4/ip_local_port_range
```

Можно также увеличить размер очереди соединений:

```
[root@caw2 ~]# echo 4096 > /proc/sys/net/ipv4/tcp_max_syn_backlog
```

Если сервер базы данных используется только локально, то можно уменьшить величину тайм-аута после закрытия сокета, в течение которого система не закрывает свою сторону соединения, страхуясь от аварийного завершения клиента. По умолчанию в большинстве систем эта величина составляет 1 минуту, что довольно долго:

```
[root@caw2 ~]# echo <значение> > /proc/sys/net/ipv4/tcp_fin_timeout
```

Как правило, для этих параметров можно оставить значения по умолчанию. Изменять их имеет смысл только тогда, когда происходит что-то экстраординарное, например, производительность сети необычайно низка или количество соединений очень велико. Поиск в Интернете по запросу TCP variables («Параметры TCP») даст обширнейший материал по этим и многим другим переменным.

Выбор операционной системы

В настоящее время для высокопроизводительных приложений MySQL чаще всего устанавливается на ОС GNU/Linux, хотя она может работать и во многих других операционных системах.

На компьютерах SPARC обычно работает система Solaris. Она же работает и на x86-оборудовании. Нередко она служит основой для приложений, требующих высокой надежности. Сложилось мнение, что с Solaris работать в некоторых отношениях сложнее, чем с GNU/Linux, но тем не менее это надежная высокопроизводительная ОС с целым рядом передовых возможностей. В частности, набирает популярность система Solaris 10. К числу ее особенностей можно отнести наличие собственной файловой системы (ZFS), разнообразие продвинутых инструментов для поиска неполадок (например, DTrace), высокопроизводительную многопоточность и технологию виртуализации Solaris Zones, упрощающую управление ресурсами.

Еще одна возможность — FreeBSD. В прошлом MySQL испытывала с ней много проблем, в основном из-за поддержки потоков, но современные версии существенно

улучшены. Сегодня не редкость встретить крупномасштабный проект с использованием MySQL, развернутый на платформе FreeBSD. ZFC доступна и на FreeBSD.

Windows обычно применяется для разработки или когда MySQL используется с приложением на настольном компьютере. Существуют корпоративные базы данных MySQL на платформе Windows, но чаще для этих целей все же используется UNIX. Не желая ввязываться в споры по поводу операционных систем, отметим, что MySQL прекрасно работает и в гетерогенной среде. Абсолютно нормально запустить сервер MySQL в системе UNIX, а веб-серверы — на платформе Windows, используя для подключения к базе высококачественный коннектор .NET, предлагаемый бесплатно вместе с MySQL. Установить соединение между клиентом в UNIX и MySQL-сервером в Windows столь же просто, как с сервером на другой UNIX-машине.

Если вы работаете на 64-разрядной аппаратной архитектуре, то выбирайте 64-разрядную операционную систему (см. подраздел «Архитектура ЦП» ранее в данной главе).

Выбор конкретного дистрибутива GNU/Linux зависит от личных предпочтений. Мы полагаем, что лучше всего остановиться на дистрибутиве, специально разработанном для серверных приложений, а не для настольных компьютеров. Принимайте во внимание время существования дистрибутива, политику выпуска версий и обновлений, проверьте, оказывает ли производитель техническую поддержку. Репутацию качественного и стабильного продукта заслужил дистрибутив Red Hat Enterprise Linux, популярен совместимый с ним на двоичном уровне (и бесплатный) дистрибутив CentOS, однако у него не самая лучшая репутация. Oracle распространяет Oracle Enterprise Linux, хорошо известны Ubuntu и Debian.

Выбор файловой системы

Выбор файловой системы зависит от выбора оперативной системы. Во многих системах, например в Windows, есть всего один или два варианта, из которых только один (NTFS) действительно заслуживает внимания. А вот GNU/Linux поддерживает множество файловых систем.

Многие хотят знать, какая файловая система обеспечивает наилучшую производительность для комбинации MySQL с GNU/Linux или даже более конкретно — какая лучше подходит для InnoDB, а какая для MyISAM. Эталонные тесты показывают, что во многих отношениях большинство файловых систем очень близки, но полагаться на файловую систему в деле повышения производительности не стоит. Производительность файловой системы сильно зависит от рабочей нагрузки, и ни одна из них не является панацеей. Как правило, каждая конкретная файловая система работает не хуже и не лучше любой другой. Исключение составляет лишь случай, когда вы приближаетесь к какому-то лимиту файловой системы, например, возникает высокая конкурентность, образуется много файлов, нарастает фрагментация и т. д.

Более важными факторами являются время восстановления после сбоя и возможность столкнуться с конкретными ограничениями, например низкая производительность при работе с каталогами, в которых содержится много файлов (этим печально

знамениты система ext2 и старые версии ext3, хотя в современных версиях ext3 и ext4 проблема решена с помощью параметра `dir_index`). Выбор файловой системы имеет первостепенное значение, когда нужна гарантия надежного хранения данных, поэтому мы настоятельно рекомендуем не экспериментировать на промышленных серверах.

По возможности предпочитайте файловые системы с журналированием, например ext3, ext4, XFS, ZFS или JFS. В противном случае проверка файловой системы после сбоя может занять много времени. Если этот аспект не очень важен, то файловые системы без журналирования могут работать несколько быстрее транзакционных. Например, ext2 в этом смысле лучше, чем ext3, хотя при желании можно отключить журналирование в ext3 командой `tunefs`. Для некоторых файловых систем имеет также значение время монтирования. Так, в многотерабайтных разделах система ReiserFS монтируется и восстанавливается довольно долго.

Для файловой системы ext3 и ее наследницы ext4 существует три варианта журналирования данных, соответствующие параметры монтирования задаются в файле `/etc/fstab`.

- `data=writeback`. Этот параметр означает, что журналируются только записи метаданных. Записи метаданных не синхронизированы с записью информации. Это самая быстрая конфигурация, и обычно она безопасна при работе с InnoDB, так как последняя ведет собственный журнал транзакций. Есть, правда, одно исключение: сбой в неподходящий момент может привести к повреждению `.frm`-файла.

  Приведем пример обстоятельств, при которых такая конфигурация может вызвать проблемы. Предположим, что программа решила расширить файл, увеличив его размер. Метаданные (размер файла) журналируются до записи самих данных в файл, который теперь стал больше. В результате в конце файла — в добавленной области — окажется мусор.

- `data=ordered`. В этом режиме тоже журналируются только метаданные, но обеспечивается хоть какая-то согласованность за счет того, что данные записываются раньше метаданных. Производительность лишь немного уступает производительности режима `writeback`, зато система ведет себя гораздо надежнее при сбоях.

  Если в этом режиме программа захочет расширить файл, то новый размер не будет отражаться в его метаданных до тех пор, пока во вновь выделенную область не будут записаны данные.

- `data=journal`. При этом значении параметра перед записью данных в окончательное место назначения производится их атомарная запись в журнал. Обычно это излишняя процедура, а накладные затраты здесь гораздо выше, чем в двух других режимах. Впрочем, в ряде случаев производительность даже увеличивается, так как наличие журнала позволяет файловой системе отложить запись данных в сам файл.

Вне зависимости от файловой системы существует ряд параметров, которые лучше отключить, поскольку они не дают никаких преимуществ, но сопряжены с накладны-

ми затратами. В связи с этим чаще всего упоминают регистрацию времени последнего доступа, потому что она подразумевает операцию записи даже в том случае, когда вы просто читаете файл. Чтобы отключить этот режим, добавьте в файл `/etc/fstab` параметры монтирования `noatime` и `nodiratime`. Иногда таким образом удается получить выигрыш в 5–10 % в зависимости от рабочей нагрузки и файловой системы (хотя в других случаях никаких ощутимых изменений не наблюдается). Приведем пример строки файла `/etc/fstab` с упомянутыми параметрами для файловой системы ext3:

```
/dev/sda2 /usr/lib/mysql ext3 noatime,nodiratime,data=writeback 0 1
```

Можно также настроить в файловой системе упреждающее чтение, поскольку иногда оно оказывается избыточным. Например, InnoDB самостоятельно прогнозирует, к каким страницам могут получать доступ в ближайшем будущем. Отключение или ограничение упреждающего чтения особенно благотворно сказывается на файловой системе UFS в Solaris. Использование флага `O_DIRECT` автоматически отключает упреждающее чтение.

Некоторые файловые системы могут не поддерживать необходимые возможности. Например, при использовании метода сброса `O_DIRECT` для InnoDB важна поддержка прямого ввода/вывода. Кроме того, одни файловые системы лучше работают с большим количеством накопителей, чем другие: так, XFS в этом отношении зачастую гораздо лучше ext3. Наконец, если вы планируете применять мгновенные снимки менеджера логических томов (LVM) для инициализации подчиненных серверов или снятия резервных копий, убедитесь, что выбранная файловая система хорошо ладит с LVM.

В табл. 9.4 обобщены характеристики некоторых наиболее часто используемых файловых систем.

Таблица 9.4. Характеристики наиболее часто используемых файловых систем

Файловая система	Операционная система	Журналирование	Большие каталоги
ext2	GNU/Linux	Нет	Нет
ext3	GNU/Linux	По желанию	По желанию/частично
ext4	GNU/Linux	Есть	Есть
HFS Plus	Mac OS	По желанию	Есть
JFS	GNU/Linux	Есть	Нет
NTFS	Windows	Есть	Есть
ReiserFS	GNU/Linux	Есть	Есть
UFS (Solaris)	Solaris	Есть	Настраивается
UFS (FreeBSD)	FreeBSD	Нет	По желанию/частично
UFS2	FreeBSD	Нет	По желанию/частично
XFS	GNU/Linux	Есть	Есть
ZFS	Solaris, FreeBSD	Есть	Есть

Обычно мы рекомендуем нашим клиентам использовать файловую систему XFS. Файловая система ext3 имеет слишком много серьезных ограничений, таких как единый мьютекс на каждый индексный дескриптор, а также применение функции `fsync()` для очистки от всех «грязных» блоков данных всей файловой системы, а не только одного файла. Файловая система ext4 появилась сравнительно недавно, поэтому к ней еще не привыкли, хотя она постепенно набирает популярность.

Выбор планировщика дисковых очередей

В GNU/Linux планировщик очередей определяет порядок, в котором запросы на устройство блочного ввода/вывода фактически отправляются на базовое устройство. По умолчанию используется «полноценная очередь», или cfq (completely fair queueing). Это нормально для повседневного использования на ноутбуках и настольных компьютерах, где помогает предотвратить нехватку ресурсов ввода/вывода, но для серверов ужасно. Это приводит к увеличению времени отклика при различных генерируемых MySQL типах рабочей нагрузки, поскольку она без необходимости стопорит некоторые запросы в очереди.

Вы можете посмотреть, какие планировщики доступны и какой активен, с помощью следующей команды:

```
$ cat /sys/block/sda/queue/scheduler
noop deadline [cfq]
```

Замените `sda` названием интересующего вас диска. В примере квадратные скобки указывают, какой планировщик используется для этого устройства. Остальные два варианта подходят для аппаратного обеспечения серверного класса и в большинстве случаев работают одинаково хорошо. Планировщик `noop` подходит для устройств, выполняющих собственное планирование за кулисами, таких как аппаратные RAID-контроллеры и SAN, а `deadline` подходит как для RAID-контроллеров, так и для подключаемых дисков. Эталонные тесты показали очень небольшую разницу между этими двумя планировщиками. Основная рекомендация такова: используйте какой-нибудь планировщик, но только не `cfq`, который может вызвать серьезные проблемы с производительностью.

Прислушиваясь к этому совету, не забывайте о толике скептицизма хотя бы потому, что планировщики дисков в разных ядрах могут быть разными, а по их названиям вы этого не поймете.

Многопоточность

MySQL использует один поток для каждого соединения. Дополнительно существуют служебные потоки, потоки специального назначения и потоки, создаваемые подсистемами хранения. В MySQL 5.5 используется плагин пула потоков произведенный Oracle, но пока неясно, насколько удобно его практическое применение.

Поэтому необходимо, чтобы операционная система эффективно поддерживала много потоков. Более того, для результативной работы с несколькими процессорами MySQL нуждается в потоках на уровне ядра, а не в адресном пространстве пользователя. Вдобавок требуется наличие эффективных примитивов синхронизации, например мьютексов. Все это должны предоставлять библиотеки, входящие в состав операционной системы.

В ОС GNU/Linux есть две библиотеки для работы с потоками: LinuxThreads и более новая Native POSIX Threads Library (NPTL). Библиотека LinuxThreads еще кое-где используется, но большинство современных дистрибутивов перешли на NPTL, а во многие дистрибутивы LinuxThreads уже не входит. Библиотека NPTL обычно потребляет меньше памяти, работает более эффективно и не страдает от множества проблем, присущих LinuxThreads.

FreeBSD также поставляется с несколькими потоковыми библиотеками. Исторически поддержка потоков в этой системе была слабой, но сейчас заметно улучшилась и на некоторых тестах даже превосходит по производительности GNU/Linux в SMP-системах. Для версии FreeBSD 6 и более поздних мы рекомендуем библиотеку libthr, а для более ранних — библиотеку linuxthreads, которая является переносом LinuxThreads на платформу FreeBSD.

В общем, теперь, когда GNU/Linux и FreeBSD получили хорошие библиотеки, проблемы с потоками ушли в прошлое.

Solaris и Windows всегда имели очень хорошую поддержку потоков. С одной оговоркой: в MyISAM до версии 5.5 потоки плохо использовались в Windows. Начиная с версии 5.5, поддержка потоков была значительно улучшена.

Подкачка

Подкачка работает тогда, когда операционная система выгружает какую-то область виртуальной памяти на диск, поскольку та не помещается в физической памяти[1]. Подкачка незаметна работающим процессам. Только операционная система знает, находится некий адрес виртуальной памяти в физической памяти или на диске.

Подкачка очень плохо отражается на производительности MySQL. Она сводит на нет весь смысл кэширования, и эффективность оказывается ниже, чем в случае, когда для кэшей отведено слишком мало памяти. В MySQL и подсистемах хранения есть немало алгоритмов, которые по-разному работают с данными, находящимися в памяти и на диске, поскольку предполагается, что доступ к хранящимся в памяти данным требует меньших затрат. Поскольку подкачка не видна пользовательским процессам, MySQL (или подсистема хранения) не знает, что данные, которые они считают находящимися в памяти, на самом деле выгружены на диск.

[1] Иногда свопинг называют подкачкой страниц (paging). Строго говоря, это разные процессы, но многие употребляют их названия как синонимы.

Это может очень сильно снизить производительность. Например, полагая, что данные все еще в памяти, подсистема хранения может захватить глобальный мьютекс (допустим, мьютекс, защищающий буферный пул в InnoDB) на время короткой операции с памятью. Но если эта операция выливается в дисковый ввод/вывод, то все остальные процессы замирают в ожидании его завершения. Следовательно, подкачка приводит к гораздо более тяжким последствиям, чем обычный ввод/вывод, выполняемый по мере необходимости.

В ОС GNU/Linux за подкачкой можно следить с помощью утилиты vmstat (в следующем разделе приведем несколько примеров). Интерес представляют столбцы `si` и `so`, отражающие динамику подкачки, а не столбец `swpd`, в котором показан объем использованного пространства в файле подкачки. Значение в столбце `swpd` может проинформировать о процессах, которые были загружены в память, но сейчас не работают, а стало быть, проблем не создают. Желательно, чтобы в столбцах `si` и `so` стояли нули, и уж точно эти показатели не должны превышать десять блоков в секунду.

В редких случаях слишком активная подкачка может привести к исчерпанию места в файле подкачки. Когда такое происходит, нехватка виртуальной памяти обычно вызывает аварийное завершение MySQL. Но даже если в файле подкачки есть место, чрезмерно интенсивная подкачка может замедлить работу операционной системы до такой степени, что невозможно будет даже войти в нее и принудительно завершить процесс MySQL. Иногда, когда заканчивается место в файле подкачки, ядро Linux может даже полностью зависнуть.

Следите за тем, чтобы виртуальная память никогда не заканчивалась! Настройте мониторинг и оповещение о том, сколько места использовано файлом подкачки. Если вы не знаете, сколько необходимо для файла подкачки, выделите для него на диске много места: он не влияет на производительность, а лишь потребляет дисковое пространство. В некоторых крупных организациях точно знают, каково будет потребление памяти, и подкачка находится под очень жестким контролем, однако в среде с несколькими многоцелевыми экземплярами MySQL, которые обслуживают переменные рабочие нагрузки, это чаще всего нецелесообразно. Если это как раз ваш случай, обязательно дайте серверу передышку, зарезервировав достаточно места для файла подкачки.

Еще одна проблема, которая часто возникает при экстремальном использовании виртуальной памяти, заключается в том, что включается убийца процессов «отсутствие памяти» (out-of-memory, OOM) и прерывает другие процессы. Часто таким процессом является MySQL, но это может быть и другой процесс, например SSH. В этом случае вы останетесь без сетевого доступа. Проблему можно предотвратить, установив значение `oom_adj` или `oom_score_adj` для SSH.

Многие проблемы, связанные с подкачкой, можно решить, правильно сконфигурировав буферы MySQL, но иногда операционная система все-таки решает выгрузить соответствующий процесс. Обычно это происходит, когда она видит, что MySQL выдает слишком много запросов на ввод/вывод, и пытается увеличить файловый кэш, чтобы он вмещал больше данных. Если памяти недостаточно, что-то приходится выгружать на диск, и этим чем-то вполне может оказаться сама MySQL. В некоторых старых версиях ядра Linux к тому же установлены контрпродуктивные приоритеты,

из-за которых выгружалось то, что выгружать было не нужно, однако впоследствии эта ошибка была исправлена.

Некоторые считают, что файл подкачки вообще следует отключить. В некоторых крайних случаях, когда иначе ядро просто отказывается вести себя прилично, это помогает, но может привести и к снижению производительности операционной системы (теоретически не должно, но на практике такое встречается). Кроме того, это попросту опасно, так как отключение подкачки означает установку жесткого ограничения на объем виртуальной памяти. Если MySQL испытывает кратковременную потребность в большом количестве памяти или на той же машине время от времени запускаются процессы, потребляющие много ресурсов (например, ночные пакетные задания), то память у MySQL может кончиться, что приведет к аварийному завершению или снятию процесса операционной системой.

Обычно операционные системы предоставляют какие-то средства контроля над виртуальной памятью и подсистемой ввода/вывода. Упомянем лишь некоторые инструменты, имеющиеся в GNU/Linux. Самый простой способ — уменьшить значение параметра `/proc/sys/vm/swappiness` до `0` или `1`. Тем самым вы говорите ядру, что оно не должно прибегать к подкачке до тех пор, пока потребность в виртуальной памяти не станет критической. Далее показано, как узнать текущее значение параметра и изменить его:

```
$ cat /proc/sys/vm/swappiness
60
```

Значение `60` является параметром подкачки по умолчанию (значение может изменяться в пределах от `0` до `100`). Это очень плохая установка по умолчанию для серверов. Она подходит только для ноутбуков. Для серверов должно быть установлено значение `0`:

```
$ echo 0 > /proc/sys/vm/swappiness
```

Другой способ заключается в том, чтобы изменить порядок чтения и записи данных подсистемой хранения. Например, установка режима `innodb_flush_method=O_DIRECT` снижает давление на подсистему ввода/вывода. Прямой ввод/вывод не кэшируется, поэтому операционная система не считает его причиной для увеличения кэша. Этот параметр применим только к InnoDB. Можно также использовать страницы большого размера, которые не выгружаются. Это работает для подсистем MyISAM и InnoDB.

Еще один способ — воспользоваться конфигурационным параметром MySQL `memlock`, который фиксирует MySQL в оперативной памяти. Такой процесс выгружаться не будет, но появится другая опасность: если невыгружаемая память кончится, то MySQL может завершиться аварийно при попытке получить дополнительную память. Проблема может возникнуть и тогда, когда невыгружаемой памяти выделяется слишком много, так что ничего не остается для самой операционной системы.

Есть немало трюков, зависящих от конкретной версии ядра, поэтому будьте осторожны, особенно при переходе на новую версию. При некоторых рабочих нагрузках бывает сложно заставить операционную систему вести себя разумно, и тогда единственный выход — уменьшить размеры буферов до условно оптимальных значений.

Состояние операционной системы

В вашей операционной системе, скорее всего, имеются инструменты, позволяющие выяснить, чем заняты сама система и оборудование. В этом разделе мы приведем примеры использования двух широко распространенных утилит — iostat и vmstat. Если в вашей системе нет какой-нибудь из них, то почти наверняка существует что-то аналогичное. Мы ставим себе целью не превратить вас в эксперта по iostat или vmstat, а просто показать, на что нужно обращать внимание, пытаясь найти проблемы с использованием подобных инструментов.

Помимо этих инструментов, в вашей операционной системе могут быть и другие, к примеру mpstat или sar. Если вас интересуют иные части системы, скажем сеть, то имеет смысл воспользоваться такими утилитами, как ifconfig (показывает, в частности, количество сетевых ошибок) или netstat.

По умолчанию vmstat и iostat выдают лишь отчет о средних значениях различных счетчиков с момента запуска сервера — это не слишком полезная информация. Но обе программы принимают в качестве аргумента интервал времени. В этом случае генерируются инкрементные отчеты, показывающие, что сервер делает в настоящий момент, а это уже куда полезнее для настройки. (В первой строке показана статистика с момент запуска системы, на нее можно не обращать внимания.)

Как интерпретировать выдачу vmstat

Давайте сначала рассмотрим пример работы vmstat. Следующая команда будет выводить отчет каждые пять секунд:

```
$ vmstat 5
procs ----------memory---------- ---swap-- -----io---- -system-- ----cpu----
r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
0  0   2632  25728  23176 740244    0    0   527   521   11    3 10  1 86  3
0  0   2632  27808  23180 738248    0    0     2   430  222   66  2  0 97  0
```

Чтобы остановить vmstat, нажмите Ctrl+C. Вид отчета зависит от операционной системы, поэтому для получения точной информации обратитесь к странице руководства.

Как упоминалось ранее, несмотря на то что мы запрашивали инкрементный отчет, в первой строке показаны средние значения за время, прошедшее с момента запуска сервера. Значения во второй строке отражают текущие показатели, а последующие строки печатаются с пятисекундным интервалом. Столбцы объединены в следующие группы.

- `procs`. В столбце `r` показано, сколько процессов ожидают выделения процессора, а в столбце `b` — сколько процессов находятся в состоянии непрерываемого сна — обычно это означает, что процесс ожидает завершения ввода/вывода (дискового, сетевого, ввода информации пользователем и т. д.).
- `memory`. В столбце `swpd` показано, сколько блоков выгружены на диск в результате страничного обмена. Оставшиеся три столбца — это количество свободных (неиспользуемых), отведенных под буферы и выделенных для кэша операционной системы блоков.

- `swap`. В столбцах из этой группы показана активность подкачки: сколько блоков в секунду подгружается с диска и выгружается на диск. Эта информация важнее, чем значение в столбце `swpd`.

  Желательно, чтобы значения в столбцах `si` и `so` оставались равными `0`, и уж точно они не должны превышать десять блоков в секунду. Кратковременные всплески активности тоже не сулят ничего хорошего.
- `io`. Значения в этих столбцах показывают, сколько блоков в секунду считывается с блочных устройств (`bi`) и записывается на них (`bo`). Обычно речь идет о дисковом вводе/выводе.
- `system`. Здесь показаны количество прерываний в секунду (`in`) и количество контекстных переключений в секунду (`cs`).
- `cpu`. Значения в этих столбцах показывают, какую часть времени (в процентах) процессор выполняет пользовательский код (вне ядра), системный код (в ядре), простаивает и ожидает завершения ввода/вывода. Если используется виртуализация, то может присутствовать и пятый столбец (`st`), показывающий, сколько времени украдено у виртуальной машины. Речь идет о том времени, в течение которого на виртуальной машине существовал процесс, готовый к исполнению, но гипервизор использовал процессор для других целей. Время, когда у виртуальной машины не было ничего готового к исполнению, из-за чего гипервизор отобрал у нее процессор, не считается украденным.

Формат отчета vmstat системно-зависимый, поэтому, если вы наблюдаете картину, отличную от описанной ранее, прочитайте страницу руководства vmstat(8). Важное замечание: величины в разделах, относящихся к памяти, подкачке и вводу/выводу, измеряются в блоках, а не в байтах. В GNU/Linux блок обычно составляет 1024 байта.

Как интерпретировать выдачу iostat

Теперь перейдем к утилите iostat[1]. По умолчанию она демонстрирует некоторые показатели работы ЦП — те же, что и vmstat. Но обычно нас интересует лишь статистика ввода/вывода, поэтому для просмотра только расширенной статистики устройств мы используем следующую команду:

```
$ iostat -dx 5
Device:  rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda         1.6     2.8 2.5 1.8  138.8   36.9     40.7      0.1  23.2   6.0   2.6
```

Как и в случае с vmstat, в первой строке показаны средние значения за период с момента запуска сервера (ее обычно опускаем для экономии места), а в последующих — инкрементные средние. В каждой строке выводятся данные об одном устройстве.

[1] Примеры, относящиеся к iostat, немного переформатированы для удобства печати. Чтобы избежать переноса строк, мы уменьшили количество знаков после запятой. Кроме того, мы показываем примеры для GNU/Linux, вывод в других операционных системах будет совершенно другим.

Существуют различные параметры, позволяющие показать или скрыть отдельные столбцы. Официальная документация немного запутанна, и нам приходилось копаться в исходном коде, чтобы выяснить, что действительно отображается. Мы вывели следующие столбцы:

- `rrqm/s` и `wrqm/s` — количество сгруппированных запросов на чтение и запись в секунду. Определение «сгруппированный» означает, что операционная система объединила несколько логических запросов в один физический запрос к устройству;
- `r/s` и `w/s` — количество запросов на чтение и запись, отправленных устройству в секунду;
- `rsec/s` и `wsec/s` — количество прочитанных и записанных секторов в секунду. В некоторых системах выводятся также столбцы `rkB/s` и `wkB/s` — количество прочитанных и записанных килобайтов в секунду. Мы их для краткости опустили;
- `avgrq-sz` — размер запроса в секторах;
- `avgqu-sz`. количество запросов, стоящих в очереди к устройству;
- `await` — количество миллисекунд, потраченных на время ожидания в очереди. К сожалению, iostat не показывает статистику отдельно для запросов на чтение и на запись, хотя они настолько различаются, что не должны усредняться вместе. Это бывает очень важно, когда вы пытаетесь обнаружить источник проблем с производительностью;
- `svctm` — количество миллисекунд, затраченных на обслуживание запросов, за исключением времени ожидания в очереди;
- `%util` — процент времени, в течение которого по крайней мере один запрос был активным. Показатель назван очень неудачно — это не использование устройства, если вы знакомы со стандартным определением использования в теории массового обслуживания. Устройство с более чем одним жестким диском (например, RAID-контроллер) должно поддерживать более высокую конкуренцию, чем 1, но значение показателя `%util` никогда не превысит 100 % (если только при его расчете не будет допущена ошибка округления). Так что этот показатель не является хорошей характеристикой насыщенности устройства, что бы ни говорилось в документации к snfv, за исключением особого случая, когда вы используете только один физический жесткий диск.

Сгенерированный отчет позволяет сделать некоторые заключения о работе подсистемы ввода/вывода. К числу наиболее важных показателей относится количество одновременно обслуживаемых запросов. Поскольку количество операций чтения и записи приведено в расчете на секунду, а время обслуживания измеряется в тысячных долях секунды, то количество запросов, одновременно обслуживаемых устройством, вычисляется по формуле[1]:

```
concurrency = (r/s + w/s) * (svctm/1000)
```

[1] По-другому охарактеризовать уровень конкурентности можно с помощью средней длины очереди, времени обслуживания и среднего времени ожидания: (avuqu_sz * svctm)/await.

Приведем пример выдачи утилиты iostat:

```
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda       105    311 298 820   3236   9052       10      127   113     9    96
```

Подставив эти значения в формулу конкурентности, получим коэффициент конкурентности, равный примерно 9,6[1]. Это означает, что в среднем за время между двумя опросами устройства одновременно обрабатывались 9,6 запроса. Данные получены для массива RAID 10 из десяти дисков, так что операционная система вполне эффективно распараллеливает запросы к этому устройству. А вот пример устройства, которое, похоже, выполняет запросы строго последовательно:

```
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sdc        81      0 280   0   3164      0       11        2     7     3    99
```

Расчет по приведенной ранее формуле показывает, что устройство обслуживает всего один запрос в секунду. Оба устройства близки к насыщению, но производительность их различна. Если вы видите, что устройство почти все время занято, как в этих примерах, то вычислите коэффициент конкурентности. Он должен быть близок к количеству физических накопителей. Если значение заметно меньше, значит, есть какие-то проблемы, например сериализация файловой системы, которую мы рассматривали ранее.

Другие полезные инструменты

Мы показали утилиты vmstat и iostat, поскольку они широкодоступны, а vmstat устанавливается по умолчанию во многих Unix-подобных операционных системах. Однако у каждого из этих инструментов есть свои ограничения, такие как непонятные единицы измерения, опросы с интервалами, которые не соответствуют интервалам обновления статистики операционной системой, и неспособность сразу увидеть все показатели. Если эти инструменты не соответствуют вашим потребностям, вас могут заинтересовать утилиты dstat (http://dag.wieers.com/home-made/dstat/) или collectl (http://collectl.sourceforge.net).

Мы также предпочитаем использовать утилиту mpstat для просмотра статистики ЦП — она позволяет получить представление о том, как ведут себя процессоры по отдельности, а не группирует их. Иногда при диагностике проблемы это очень важно. Возможно, вы обнаружите, что утилита blktrace будет полезна при анализе использования дискового ввода/вывода.

Мы написали собственную замену утилите iostat — инструмент под названием pt-diskstats. Он входит в пакет Percona и является нашим ответом на некоторые недостатки iostat, такие как, например, представление данных по чтению и записи в совокупности и отсутствие наглядности по конкурентности. Он интерактивен и управляется с помощью клавиш, поэтому вы можете увеличивать и уменьшать масштаб, изменять агрегированность, отфильтровывать устройства, показывать и скрывать столбцы. Это отличный способ проанализировать вдоль и поперек выборку статистики диска,

[1] Если вы сами выполните расчеты, вы получите результат около 10, поскольку мы округлили вывод утилиты iostat для удобства печати. Поверьте, результат действительно 9,6.

которую вы можете собрать с помощью простого сценария оболочки, даже если у вас не установлен наш инструмент. Вы можете зафиксировать образцы активности диска и отправить их по электронной почте или сохранить для последующего анализа. Фактически инструменты pt-stalk, pt-collect и pt-sift, о которых мы рассказали в главе 3, созданы для хорошей совместной работы с pt-diskstats.

Машина с нагруженным процессором

Если процессор сильно нагружен, то vmstat обычно показывает в столбце `us` большое значение — долю времени, в течение которого процессор занят выполнением пользовательского кода.

Кроме того, большое значение может быть в столбце `sy`, который характеризует использование ЦП системой, — увидев значение выше 20 %, может начинать волноваться. В большинстве случаев вы увидите также, что в очереди к ЦП стоят несколько процессов (столбец `r`). Приведем пример:

```
$ vmstat 5
procs -----------memory----------   ---swap-- -----io---- --system-- ----cpu----
 r  b   swpd   free   buff  cache     si   so    bi    bo   in    cs us sy id wa
10  2 740880  19256  46068 13719952    0    0  2788 11047 1423 14508 89  4  4  3
11  0 740880  19692  46144 13702944    0    0  2907 14073 1504 23045 90  5  2  3
 7  1 740880  20460  46264 13683852    0    0  3554 15567 1513 24182 88  5  3  3
10  2 740880  22292  46324 13670396    0    0  2640 16351 1520 17436 88  4  4  3
```

Обратите внимание также на большое количество контекстных переключений (столбец `cs`), хотя не стоит об этом беспокоиться, пока значение менее 100 000 в секунду. *Контекст переключается*, когда операционная система приостанавливает один процесс и замещает его другим.

Например, запрос, выполняющий сканирование не покрывающего индекса в таблице MyISAM, будет считывать запись из индекса, а затем читать строку со страницы на диске. Если страница не находится в кэше операционной системы, с диска будет производиться физическое чтение, из-за чего переключатель контекста приостановит процесс до завершения ввода/вывода. Такой запрос может вызвать множество переключателей контекста.

Запустив на той же машине iostat (верхняя строка, где размещаются средние значения за все время с момента запуска, опущена), мы увидим, что коэффициент загруженности диска — менее 50 %:

```
$ iostat -dx 5
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda           0   3859  54 458   2063  34546       71        3     6     1    47
dm-0          0      0  54 4316  2063  34532        8       18     4     0    47
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm util
sda           0   2898  52 363   1767  26090       67        3     7     1    45
dm-0          0      0  52 3261  1767  26090        8       15     5     0    45
```

Основное время эта машина тратит на процессорные операции, однако объем ввода/вывода также велик, что вполне нормально для серверов баз данных. В то же время

типичный веб-сервер потребляет очень много ресурсов ЦП, но очень слабо загружает подсистему ввода/вывода, поэтому для веб-сервера картина будет отличаться от показанной.

Машина с нагруженной подсистемой ввода/вывода

Если рабочая нагрузка сопряжена преимущественно с вводом/выводом, то ЦП проводит много времени в ожидании завершения запросов ввода/вывода. Это означает, что vmstat покажет много процессов в состоянии непрерываемого сна (столбец `b`), значение в столбце `wa` также будет велико, например:

```
$ vmstat 5
procs -----------memory----------   ---swap-- -----io---- --system-- ----cpu----
r  b   swpd   free   buff  cache     si   so    bi    bo   in    cs us sy id wa
5  7 740632  22684  43212 13466436    0    0  6738 17222 1738 16648 19  3 15 63
5  7 740632  22748  43396 13465436    0    0  6150 17025 1731 16713 18  4 21 58
1  8 740632  22380  43416 13464192    0    0  4582 21820 1693 15211 16  4 24 56
5  6 740632  22116  43512 13463484    0    0  5955 21158 1732 16187 17  4 23 56
```

Вывод iostat демонстрирует, что на этой машине диски полностью загружены[1]:

```
$ iostat -dx 5
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda           0   5396 202  626  7319  48187       66       12    14     1   101
dm-0          0      0 202 6016  7319  48130        8       57     9     0   101
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda           0   5810 184  665  6441  51825       68       11    13     1   102
dm-0          0      0 183 6477  6441  51817        8       54     7     0   102
```

Величина `%util` может быть больше 100 % из-за ошибок округления.

Когда можно считать, что подсистема ввода/вывода сильно нагружена? Если емкости буферов хватает для обслуживания запросов на запись, то обычно (но не всегда) это означает, что диски не справляются с запросами на чтение, даже если выполняется очень много операций записи. Кажется, что это противоречит ожиданиям, но давайте задумаемся о природе чтения и записи.

Запросы на запись могут быть либо буферизованными, либо синхронными. Как мы уже отмечали, буферизация возможна на различных уровнях, таких, как операционная система, RAID-контроллер и т. д.

Запросы на чтение по природе своей синхронны. Программа, конечно, может предположить, что в ближайшем будущем потребуются некоторые данные, и отправить асинхронный запрос на упреждение (упреждающее чтение). Однако чаще она обнаруживает, что некоторые данные необходимы, и без них не может продолжать работу. Поэтому чтение обязано быть синхронным: процесс блокируется до завершения запроса.

---

[1] Во втором издании этой книги мы объединили понятия «всегда занятые» и «полностью насыщенные». Диски, всегда что-то делающие, не обязательно используются без остатка, поскольку они могут также поддерживать некоторую конкурентность.

Взгляните на ситуацию следующим образом: вы можете отправить запрос на запись, который сохраняется в каком-то буфере и выполняется позже. Можно даже отправить много таких запросов в течение одной секунды. Если буфер работает корректно и в нем достаточно места, то каждый запрос завершается очень быстро, а реальные операции записи на физический диск впоследствии можно будет сгруппировать и выполнить в другом порядке для повышения эффективности.

Но с чтением так поступить нельзя — неважно, короткий запрос или длинный, диск не может ответить: «Вот твои данные, а прочту я их попозже». Поэтому ожидание ввода/вывода вызвано, как правило, операциями чтения.

Машина с интенсивной подкачкой

Если машина активно выгружает и подгружает данные, значение в столбце `swpd` может быть как большим, так и маленьким. Однако не следует допускать, чтобы значения в столбцах `si` и `so` были велики. Вот как может выглядеть выдача vmstat на машине с интенсивной подкачкой:

```
$ vmstat 5
procs -----------memory------------ ---swap---- -----io---- --system-- ----cpu----
r  b    swpd   free   buff    cache   si   so    bi    bo   in    cs us sy id wa
0 10 3794292  24436  27076 14412764 19853  9781 57874  9833 4084  8339  6 14 58 22
4 11 3797936  21268  27068 14519324 15913 30870 40513 30924 3600  7191  6 11 36 47
0 37 3847364  20764  27112 14547112   171 38815 22358 39146 2417  4640  6  8  9 77
```

Простаивающая машина

Для полноты картины приведем выдачу vmstat, полученную на простаивающей машине. Обратите внимание на то, что здесь нет ни готовых к выполнению, ни блокированных процессов, а столбец `idle` показывает, что процесс простаивает 100 % времени. Эти данные были получены на компьютере под управлением Red Hat Enterprise Linux 5, в них присутствует столбец `st`, в котором отображается время, украденное у виртуальной машины:

```
$ vmstat 5
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy  id wa st
0  0    108 492556   6768 360092    0    0   345   209    2   65  2  0  97  1  0
0  0    108 492556   6772 360088    0    0     0    14  357   19  0  0 100  0  0
0  0    108 492556   6776 360084    0    0     0     6  355   16  0  0 100  0  0
```

Итоги главы

Выбор и конфигурация оборудования для MySQL и конфигурация MySQL для аппаратного обеспечения — это не тайное искусство. В целом вам требуются те же знания и умения, что и для достижения большинства других целей. Однако существуют кое-какие специфические особенности конфигурации MySQL, которые стоит знать.

Обычно большинству людей мы советуем найти хороший баланс между производительностью и стоимостью. Прежде всего, нам нравится использовать стандартные серверы по многим причинам. Например, если у вас возникли проблемы с сервером и нужно выключить его на время диагностики или вы в рамках диагностики просто хотите попробовать заменить его на другой сервер, то сделать это намного проще с сервером за 5000 долларов, чем с тем, который стоит 50 000 долларов или больше. Также MySQL, как правило, лучше подходит — как с точки зрения самого программного обеспечения, так и с точки зрения типичных рабочих нагрузок — для стандартного оборудования.

Четыре основных ресурса, которые нужны MySQL, — это ЦП, память, диск и сетевые ресурсы. Сеть в отличие от процессора, памяти и дисков, как правило, не оказывается серьезным узким местом. Обычно вам требуется много быстрых процессоров для MySQL, хотя, если нужно выбирать между количеством и быстродействием, при прочих равных условиях выбирайте быстродействие.

Взаимоотношения между процессорами, памятью и дисками запутанны, и часто проблемы, возникшие в одной области, проявляются совсем в другом месте. Прежде чем бросать ресурсы на решение проблемы, спросите себя, не следует ли бросить их совсем в другое место. Если диски сильно нагружены, решите, нужно ли вам больше мощностей для ввода/вывода или просто больше памяти? Ответ зависит от размера рабочего множества, представляющего собой набор данных, который требуется наиболее часто в течение заданного времени.

На момент написания книги мы считаем целесообразным действовать следующим образом. Не стоит использовать более двух сокетов. Даже двухсокетная система может предложить множество ядер процессора и аппаратных потоков, а процессоры, доступные для четырех сокетов, значительно дороже. Кроме того, они не так широко используются (и, следовательно, хуже протестированы и менее надежны). Кроме того, у них более низкие тактовые частоты. Наконец, в четырехсокетных системах, по-видимому, большие затраты на кросс-сокетную синхронизацию. Что касается памяти, то нам нравится заполнять наши серверы недорогой памятью серверного класса. В настоящий момент на многих стандартных серверах имеется 18 DIMM слотов и 8 Гбайт DIMM. Это хороший размер — их цена за гигабайт такая же, как и у меньших DIMM, но гораздо меньше, чем у 16 Гбайт DIMM. Именно поэтому в настоящее время часто встречаются серверы с 144 Гбайт памяти. Эта тенденция со временем изменится — самыми выгодными в итоге станут 16 Гбайт DIMM, возможно, будет другое количество слотов в формфакторах стандартных серверов, но общий принцип, вероятно, сохранится.

Выбор устройства долговременного хранения, по существу, сводится к трем вариантам. Перечислим их в порядке возрастания производительности: SAN, обычные диски и твердотельные устройства. Опишем разницу в двух словах.

- SAN-сети стоит применять, когда вам нужны их возможности и мощность. Они хорошо работают при больших рабочих нагрузках, но дороги. Кроме того, у них большая задержка при небольших операциях произвольного ввода/вывода, особенно когда используется более медленное соединение, такое как NFS, или

рабочее множество не помещается во внутренний кэш сетей SAN. Остерегайтесь неожиданностей с производительностью сетей SAN и тщательно планируйте свои действия на случай реализации негативных сценариев.

- Обычные диски большие, дешевые и медленные при произвольном чтении. Для большинства сценариев лучшим выбором являются тома RAID 10 для серверных дисков. Обычно целесообразно использовать аппаратный RAID-контроллер с блоком резервного питания аккумулятора и кэшем записи, для которого установлена политика WriteBack. Такая конфигурация должна работать очень хорошо при большинстве задаваемых вами нагрузок.
- Твердотельные хранилища данных относительно малы и дороги, но они очень быстро работают при произвольном вводе/выводе. Существует два класса: SSD и PCIe. Грубо говоря, SSD дешевле, медленнее и менее надежны. Для повышения их долговечности необходим RAID, однако большинство аппаратных RAID-контроллеров не справляются с этой задачей. Устройства PCIe стоят дорого и имеют ограниченную емкость, но они чрезвычайно быстры и надежны и им не нужен RAID.

Твердотельные устройства отлично подходят для повышения производительности сервера в целом, а иногда недорогой SSD — это способ обработки определенной рабочей нагрузки, которая плохо реализуется на обычных дисках, например репликации. Если действительно нужна мощность, то потребуется устройство PCIe. Добавление серверу быстрого ввода/вывода приводит к перемещению узкого места на процессор, а иногда и в сеть.

MySQL и InnoDB не могут в полной мере воспользоваться преимуществами, предлагаемыми высококачественными твердотельными хранилищами, а в некоторых случаях это неспособны сделать и операционные системы. Однако эта ситуация улучшается довольно быстро. В Percona Server реализовано много усовершенствований, касающихся твердотельных накопителей. Кроме того, много их ожидается в готовящемся релизе MySQL 5.6.

Что касается операционной системы, есть всего несколько важных вещей, которые вам нужно делать правильно, в основном они связаны с хранением, сетью и управлением виртуальной памятью. Если вы, как и большинство пользователей MySQL, работаете с GNU/Linux, советуем применять файловую систему XFS и задать для подкачки и планировщика очередей значения, подходящие для сервера. Возможно, вам потребуется изменить некоторые параметры сети, а изменение других установок (например, отключение SELinux) зависит от ваших предпочтений.

10 Репликация

Встроенные в MySQL средства репликации являются основой для построения крупных высокопроизводительных приложений и используют так называемую горизонтально масштабируемую архитектуру. Они позволяют сконфигурировать один или несколько серверов в качестве подчиненных другому серверу (такие серверы называют *репликами*[1]). При этом данные реплики синхронизированы с данными главного сервера. Это полезно не только при создании высокопроизводительных приложений — репликация является краеугольным камнем многих стратегий высокой доступности, масштабируемости, аварийного восстановления, резервного копирования, анализа, хранения данных и многих других задач. Фактически масштабируемость и высокая доступность — это связанные темы, которые мы рассмотрим в этой и следующих двух главах.

В текущей главе поговорим обо всех аспектах репликации. Начнем с обзора принципов работы, затем перейдем к простейшей настройке сервера, далее продемонстрируем более сложные конфигурации и расскажем о том, как управлять реплицированными серверами и оптимизировать их. Хотя эта книга посвящена прежде всего вопросам производительности, для репликации не менее важны корректность и надежность, поэтому мы остановимся на том, с какими проблемами можно столкнуться в ходе репликации и как правильно ее организовать.

Обзор репликации

Основная задача, которую призвана решить репликация, — это синхронизация данных одного сервера с данными другого. К одному главному серверу можно подключить несколько подчиненных, которые будут синхронизированы с ним, причем подчиненный сервер может, в свою очередь, выступать в роли главного. Сеть, состоящая из главных и подчиненных серверов, может быть организована разными способами.

MySQL поддерживает две разновидности репликации: *покомандную* и *построчную*. Покомандная (или логическая) репликация существует со времен версии 3.23,

[1] Возможно, вам встречался термин «раб» (slave). Мы постараемся его избегать.

в настоящее время именно она обычно используется в устройствах, находящихся в промышленной эксплуатации. Построчная репликация появилась в версии MySQL 5.1. В обоих случаях изменения записываются в двоичный журнал[1] на главном сервере и воспроизводятся на подчиненном, причем оба варианта асинхронны, то есть не гарантируется, что копия данных на подчиненном сервере актуальна в любой момент. Относительно величины отставания также не дается никаких гарантий. Если запросы сложны, то подчиненный сервер может отставать от главного на секунды, минуты и даже часы.

Репликация в MySQL в основном обратно совместима. Это означает, что сервер с более поздней версией MySQL может быть подчинен серверу, на котором установлена ранняя версия. Однако старые версии обычно не могут выступать в роли подчиненных для более свежих — они не распознают новые возможности, синтаксические конструкции SQL, да и форматы файлов репликации могут различаться. Например, невозможно реплицировать главный сервер версии MySQL 5.1 на подчиненный версии 4.0. Мы рекомендуем протестировать схему репликации до перехода на новую версию, в которую были внесены серьезные изменения, например при переходе с 4.1 на 5.0 или с 5.1 на 5.5. Обновления в рамках вспомогательных версий, например с 5.1.51 до 5.1.58, обычно совместимы — почитайте журнал изменений, чтобы узнать, что конкретно изменилось при переходе на другую версию. Вообще говоря, репликация не сильно увеличивает накладные затраты на главном сервере. Правда, на нем требуется включить двоичный журнал, что само по себе весьма ощутимо, но это нужно сделать, если вы хотите снимать нормальные резервные копии. Помимо записи в двоичный журнал, небольшую нагрузку (в основном на сеть) дает добавление каждого подчиненного сервера. Если реплики считывают старые двоичные журналы главного сервера, а не просто реагируют на последние событиями, накладные затраты могут быть намного выше из-за операций ввода/вывода, необходимых для чтения старых журналов. Этот процесс может вызывать также конкуренцию за мьютекс, которая помешает транзакции. Наконец, если вы копируете очень высокую пропускную нагрузку (скажем, 5000 или более транзакций в секунду) на много подчиненных серверов, накладные затраты, связанные с пробуждением потоков всех реплик для отправки им событий, будут суммироваться.

Репликация применима для масштабирования операций чтения, которые можно адресовать подчиненному серверу, но для масштабирования записи она не очень подходит, если только не учесть это требование при проектировании системы. Подключение большого количества подчиненных серверов просто приводит к тому, что запись выполняется многократно, по одному разу на каждом из них. Система в целом ограничена количеством операций записи, которые может выполнить самое слабое ее звено.

При наличии нескольких подчиненных серверов репликация становится дорогим удовольствием, так как данные без нужды дублируются несколько раз. Например,

[1] Информацию о двоичном журнале можете найти в главе 8, далее в этой главе и в главе 15.

если к одному главному серверу подключены десять подчиненных, то на главном сервере оказывается 11 одинаковых копий данных, которые дублируются в 11 разных кэшах. Это можно считать аналогом RAID 1 с 11 дисками. Подобное использование оборудования неэкономно, тем не менее такая конфигурация встречается на удивление часто. В дальнейшем мы обсудим различные способы сгладить эту проблему.

Проблемы, решаемые репликацией

Перечислим несколько типичных случаев применения репликации.

- *Распространение данных.* Обычно на репликацию в MySQL расходуется не очень большая часть пропускной способности сети, хотя, как мы увидим позже, построчная репликация, представленная в MySQL 5.1, может потребовать гораздо большей пропускной способности, чем более традиционная покомандная. К тому же ее можно в любой момент остановить и затем возобновить. Это полезно, если копия данных хранится в территориально удаленном пункте, например в другом центре обработки данных. Удаленный подчиненный сервер может работать даже с непостоянным (намеренно или по другим причинам) соединением. Но если вы хотите обеспечить минимальное отставание реплики, то следует использовать надежный канал с малым временем задержки.
- *Балансировка нагрузки.* С помощью репликации можно распределить запросы на чтение между несколькими серверами MySQL, что особенно продуктивно в приложениях с интенсивным чтением. Реализовать несложную балансировку нагрузки можно, внеся совсем немного изменений в код. Для небольших приложений достаточно просто зашить в программу несколько доменных имен или воспользоваться циклическим разрешением DNS-имен (когда с одним доменным именем связаны несколько IP-адресов). Возможны и более изощренные решения. Стандартные технологии балансировки нагрузки, в частности сетевые балансировщики, прекрасно послужат для распределения нагрузки между несколькими серверами MySQL. Неплохо зарекомендовал себя и проект Linux Virtual Server (LVS). Подробнее о балансировке нагрузки поговорим в главе 11.
- *Резервное копирование.* Репликация — это большое подспорье для резервного копирования. Однако подчиненный сервер все же не может использоваться в качестве резервной копии и не способен заменить настоящее резервное копирование.
- *Высокая доступность и аварийное переключение на резервный сервер.* Репликация позволяет исправить ситуацию, при которой сервер MySQL является единственной точкой отказа приложения. Хорошая система аварийного переключения, имеющая в составе реплицированные подчиненные серверы, способна существенно сократить время простоя при отказе. Мы рассмотрим эту тему в главе 12.
- *Тестирование новых версий MySQL.* Очень часто на подчиненный сервер устанавливают новую версию MySQL и перед тем, как обновлять все серверы, проверяют, все ли запросы работают нормально.

Как работает репликация

Перед тем как вплотную заняться настройкой репликации, посмотрим, как же на самом деле MySQL реплицирует данные. В самом общем виде репликацию можно описать как процедуру, состоящую из трех шагов.

1. Главный сервер записывает изменения данных в двоичный журнал. Эти записи называются событиями двоичного журнала.
2. Подчиненный сервер копирует события двоичного журнала в свой журнал ретрансляции.
3. Подчиненный сервер воспроизводит события из журнала ретрансляции, применяя изменения к собственным данным.

Это лишь общая картина, в действительности каждый шаг весьма сложен. На рис. 10.1 приведена более подробная схема процедуры репликации.

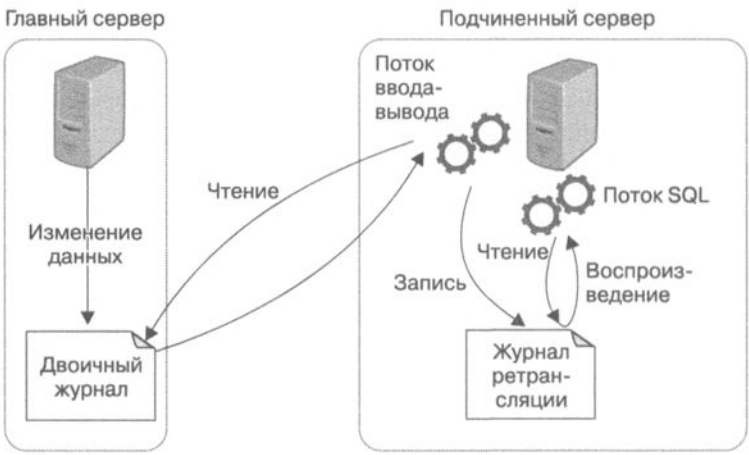
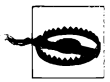


Рис. 10.1. Как работает репликация в MySQL

Первый этап данного процесса — запись в двоичный журнал на главном сервере (как ее настроить, мы объясним чуть позже). Непосредственно перед тем, как завершить транзакцию, обновляющую данные, главный сервер заносит изменения в свой двоичный журнал. MySQL записывает транзакции последовательно, даже если во время выполнения перемежаются команды из разных транзакций. Записав события в двоичный журнал, главный сервер просит подсистему хранения выполнить коммит транзакции.

На следующем этапе подчиненный сервер копирует двоичный журнал главного сервера на свой жесткий диск, в так называемый *журнал ретрансляции*. Первым делом он запускает *подчиненный поток ввода/вывода*. Этот поток открывает обычное клиентское соединение с главным сервером, а затем запускает специальный процесс *дампа двоичного журнала* (соответствующей команды в языке SQL не существует). Этот процесс читает события из двоичного журнала главного сервера. Он не опрашивает события активно. Обнаружив конец журнала, процесс выгрузки дампа засыпает и ждет, пока главный сервер не просигнализирует о появлении новых событий.

Прочитанные события поток ввода/вывода записывает в журнал ретрансляции на подчиненном сервере.



До выхода версии MySQL 4.0 репликация во многих отношениях работала по-другому. Например, первоначально никакого журнала ретрансляции не было, поэтому для репликации использовались два, а не три потока. Но сейчас, как правило, эксплуатируются более поздние версии сервера, поэтому рассказывать об уже неактуальных деталях мы не будем.

На последнем этапе в дело вступает *подчиненный поток SQL*. Он читает и воспроизводит события из журнала ретрансляции, приводя данные на подчиненном сервере в соответствие с главным сервером. Если поток SQL успевает за потоком ввода/вывода, журнал ретрансляции обычно остается в кэше операционной системы, так что накладные затраты на работу с этим журналом очень низкие. События, исполняемые потоком SQL, могут записываться также в собственный двоичный журнал подчиненного сервера, что бывает полезно при реализации отдельных сценариев, которые мы рассмотрим далее в этой главе.

На рис. 10.1 показаны только два потока репликации на подчиненном сервере, но есть и еще один поток на главном сервере: как и при других соединениях с сервером MySQL, соединение, открытое подчиненным сервером, запускает поток на главном.

Такая архитектура репликации позволяет отсоединить друг от друга процессы выборки и воспроизведения событий на подчиненном сервере и сделать их асинхронными. Иными словами, поток ввода/вывода может работать независимо от потока SQL. Кроме того, она накладывает на процедуру репликации определенные ограничения, среди которых важнее всего то, что *репликация сериализуется на подчиненном сервере*. Это означает, что обновления, реализованные на главном сервере, возможно, параллельно (в разных потоках), на подчиненном сервере распараллелены быть не могут, поскольку выполняются в одном потоке. Как мы увидим в дальнейшем, при некоторых характеристиках рабочей нагрузки это может стать узким местом. Хотя существует ряд решений, позволяющих обойти это ограничение, большинство пользователей по-прежнему используют один поток.

Настройка репликации

В MySQL настройка репликации не вызывает особых сложностей, но у основных шагов есть много вариаций, зависящих от конкретного сценария. Самый простой случай — когда главный и подчиненный серверы только что установлены и еще не введены в эксплуатацию. На верхнем уровне процедура выглядит следующим образом.

1. Завести учетные записи репликации на каждом[1] сервере.
2. Сконфигурировать главный и подчиненный серверы.

[1] Это не обязательно, но мы рекомендуем так сделать; причины объясним чуть позже.

3. Скомандовать подчиненному серверу, чтобы он соединился с главным и начал реплицировать данные с него.

Здесь подразумевается, что многие принимаемые по умолчанию параметры удовлетворительны, и это действительно так, если главный и подчиненный серверы только что установлены и обладают одними и теми же данными (стандартной базой данных `mysql`). Мы последовательно опишем действия на каждом шаге, предполагая, что серверы называются `server1` (IP-адрес 192.168.0.1) и `server2` (IP-адрес 192.168.0.2). Затем объясним, как инициализировать подчиненный сервер с помощью уже эксплуатируемого, и подробно рассмотрим рекомендуемую конфигурацию репликации.

Создание аккаунтов репликации

В MySQL предусмотрено несколько специальных привилегий, необходимых для запуска репликации. Подчиненный поток ввода/вывода, работающий на подчиненном сервере, устанавливает TCP/IP-соединение с главным сервером. Это означает, что на главном сервере должен быть создан аккаунт, наделенный такими привилегиями, чтобы поток ввода/вывода мог соединиться от его имени и читать двоичный журнал главного сервера. Далее показано, как создать такой аккаунт — мы назвали его `repl`:

```
mysql> GRANT REPLICATION SLAVE, REPLICATION CLIENT ON *.*
    -> TO repl@'192.168.0.%' IDENTIFIED BY 'p4ssword';
```

Такой аккаунт создается как на главном, так и на подчиненном сервере. Отметим, что мы разрешили пользователю устанавливать соединение только из локальной сети, потому что аккаунт репликации имеет возможность читать все изменения на сервере, что делает его привилегированной учетной записью. (Несмотря на то что у него нет возможности выполнять `SELECT` или изменять данные, он все равно может видеть некоторые данные в двоичных журналах.)

Учетной записи репликации на самом деле нужна только привилегия REPLICATION SLAVE на главном сервере, а REPLICATION CLIENT не нужна ни на главном, ни на подчиненном сервере. Так зачем же мы их дали на обоих серверах? Тому есть две причины.

- Аккаунту, который применяется для мониторинга репликации и управления ею, нужна привилегия REPLICATION CLIENT, и лучше не усложнять себе жизнь, а использовать одну и ту же запись для обеих целей, а не создавать отдельный аккаунт пользователя для этой цели.
- Если вы заведете учетную запись на главном сервере, а затем клонируете его для настройки подчиненного сервера, то подчиненный сервер уже будет подготовлен для роли главного на случай, если вы захотите поменять серверы ролями.

Конфигурирование главного и подчиненного серверов

Следующий шаг — включение нескольких параметров на главном сервере, в роли которого будет выступать `server1`. Необходимо включить двоичный журнал и задать идентификатор сервера. Введите (или убедитесь в наличии) такие строки в файл `my.cnf` на главном сервере:

```
log_bir   = mysql-bin
server_id = 10
```

Конкретные значения выберите по своему усмотрению. Мы пошли по простейшему пути, но вам, возможно, захочется чего-то более хитрого.

Необходимо явно назначить серверу уникальный идентификатор. Мы задали `10`, а не `1`, так как значение `1` сервер обычно выбирает по умолчанию, если не задано другое (это зависит от версии — некоторые версии MySQL в этом случае вообще не работают). Поэтому выбор `1` легко может вызвать конфликты между серверами, которым идентификатор явно не назначен. Часто в качестве идентификатора выбирают последний октет IP-адреса сервера, предполагая, что он уникален и в будущем не изменится (то есть все серверы находятся в одной подсети). Стоит выбрать какое-то соглашение, которое кажется вам разумным, и придерживаться его.

Если двоичный журнал ранее не был включен на главном сервере, то MySQL придется перезапустить. Чтобы проверить, создан ли на главном сервере двоичный журнал, выполните команду `SHOW MASTER STATUS` и сравните ее результат с приведенным далее. MySQL добавляет к имени файла несколько цифр, поэтому истинное имя будет отличаться от заданного вами:

```
mysql> SHOW MASTER STATUS;
+------------------+----------+--------------+------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+------------------+----------+--------------+------------------+
| mysql-bin.000001 |       98 |              |                  |
+------------------+----------+--------------+------------------+
1 row in set (0.00 sec)
```

Файл `my.cnf` на подчиненном сервере выглядит примерно так же, как на главном, но с некоторыми дополнениями. Подчиненный сервер также необходимо перезапустить:

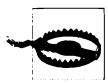
```
log_bin           = mysql-bin
server_id         = 2
log_bin           = mysql-bin
server_id         = 2
relay_log         = /var/lib/mysql/mysql-relay-bin
log_slave_updates = 1
read_only         = 1
```

Некоторые из этих параметров, строго говоря, необязательны, а для других мы явно задали значения, совпадающие со значениями по умолчанию. На самом деле

на подчиненном сервере обязательным является только параметр `server_id`, но мы включили также `log_bin` и присвоили файлу журнала явное имя. По умолчанию имя этого файла совпадает с именем хоста, но при такой конфигурации могут возникнуть проблемы, если в будущем имя хоста изменится. Для простоты мы назвали главный и подчиненный серверы одинаково, но вы можете сделать иначе.

Мы добавили также два необязательных параметра: `relay_log` (определяет имя и местоположение журнала ретрансляции) и `log_slave_updates` (обязывает подчиненный сервер записывать реплицированные события в собственный двоичный журнал). Последнее добавляет подчиненному серверу работы, но, как мы вскоре убедимся, имеются обоснованные причины для того, чтобы задавать эти необязательные параметры на всех подчиненных серверах.

Некоторые включают только двоичный журнал, не задавая параметр `log_slave_updates`, чтобы сразу же увидеть, изменяются ли какие-нибудь данные на подчиненном сервере, например, из-за неправильно сконфигурированного приложения. Если возможно, то лучше использовать параметр `read_only`, который не дает модифицировать данные никому, кроме потоков со специальными привилегиями (не давайте пользователям больше привилегий, чем реально необходимо для работы!). Однако часто параметр `read_only` оказывается непрактичным, особенно если некое приложение должно иметь возможность создавать таблицы на подчиненных серверах.



Не помещайте такие конфигурационные параметры, как master_ host и master_port, в файл my.cnf на подчиненном сервере. Это устаревший способ конфигурирования подчиненного сервера. Он может создать проблемы и не дает никаких преимуществ.

Запуск подчиненного сервера

Следующий шаг — сообщить подчиненному серверу о том, как соединиться с главным и начать воспроизведение двоичных журналов. Для этой цели используется не файл `my.cnf`, а команда `CHANGE MASTER TO`. Она полностью заменяет соответствующие настройки в файле `my.cnf`. Кроме того, она позволяет впоследствии указать подчиненному серверу другой главный без перезапуска. Далее приведена простейшая форма команды, необходимой для запуска репликации на подчиненном сервере:

```
mysql> CHANGE MASTER TO MASTER_HOST='server1',
    -> MASTER_USER='repl',
    -> MASTER_PASSWORD='p4ssword',
    -> MASTER_LOG_FILE='mysql-bin.000001',
    -> MASTER_LOG_POS=0;
```

Параметр MASTER_LOG_POS устанавливается в 0, потому что это начало журнала. После того как эта команда отработает, выполните команду SHOW SLAVE STATUS и проверьте, правильно ли установлены параметры подчиненного сервера:

```
mysql> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
             Slave_IO_State:
                Master_Host: server1
                Master_User: repl
                Master_Port: 3306
              Connect_Retry: 60
            Master_Log_File: mysql-bin.000001
        Read_Master_Log_Pos: 4
             Relay_Log_File: mysql-relay-bin.000001
              Relay_Log_Pos: 4
      Relay_Master_Log_File: mysql-bin.000001
           Slave_IO_Running: No
          Slave_SQL_Running: No
                            ...опущено...
      Seconds_Behind_Master: NULL
```

Столбцы Slave_IO_State, Slave_IO_Rurrirg и Slave_SQL_Running показывают, что процессы репликации на подчиненном сервере не запущены. Внимательный читатель заметит также, что позиция указателя журнала равна 4, а не 0. Это объясняется тем, что 0 — это не столько истинное значение указателя, сколько признак «в начале файла журнала». MySQL знает, что данные первого события начинаются в позиции 4[1].

Чтобы запустить репликацию, выполните следующую команду:

```
mysql> START SLAVE;
```

Она не должна выводить никаких сообщений, включая сообщения об ошибках. Снова проверьте состояние подчиненного сервера командой SHOW SLAVE STATUS:

```
mysql> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
             Slave_IO_State: Waiting for master to send event
                Master_Host: server1
                Master_User: repl
                Master_Port: 3306
              Connect_Retry: 60
            Master_Log_File: mysql-bin.000001
        Read_Master_Log_Pos: 164
             Relay_Log_File: mysql-relay-bin.000001
              Relay_Log_Pos: 164
      Relay_Master_Log_File: mysql-bin.000001
           Slave_IO_Running: Yes
          Slave_SQL_Running: Yes
                            ...опущено...
      Seconds_Behind_Master: 0
```

[1] В действительности из результата приведенной ранее команды SHOW MASTER STATUS следует, что указатель находится в позиции 98. Подчиненный сервер (s/slave/replica/) разберется в этом, когда установит соединение с главным, чего пока не произошло.

Обратите внимание на то, что теперь на подчиненном сервере работают потоки ввода/вывода и SQL, а переменная состояния `Seconds_Behind_Master` отлична от `NULL` (что она означает, мы расскажем позже). Поток ввода/вывода ожидает события от главного сервера, то есть к настоящему моменту он прочитал из двоичного журнала все записи, которые там были. Позиции указателей в обоих журналах сместились, иными словами, какие-то события были прочитаны и обработаны (на вашем сервере картина может быть иной). Если сейчас выполнить какое-нибудь изменение на главном сервере, то указатели позиций на подчиненном увеличатся. Кроме того, изменения будут применены к подчиненному серверу!

Потоки репликации должны быть видны в списках процессов на главном и подчиненном серверах. На главном вы увидите соединение, созданное потоком ввода/вывода, который работает на подчиненном сервере:

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 55
   User: repl
   Host: replica1.webcluster_1:54813
     db: NULL
Command: Binlog Dump
   Time: 610237
  State: Has sent all binlog to slave; waiting for binlog to be updated
   Info: NULL
```

На подчиненном сервере должно быть два потока — ввода/вывода и SQL:

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 1
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 611116
  State: Waiting for master to send event
   Info: NULL
*************************** 2. row ***************************
     Id: 2
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 33
  State: Has read all relay log; waiting for the slave I/O thread to update it
  Info: NULL
```

Приведенные примеры распечаток получены на серверах, которые проработали довольно долго, поэтому в столбце `Time` для потока ввода/вывода на главном и подчиненном серверах выводится большое значение. SQL-поток на подчиненном сервере простаивает 33 секунды, то есть в течение этого времени не было воспроизведено ни одного события.

Эти процессы всегда работают от имени учетной записи `system user`, но значения в остальных столбцах могут отличаться от показанных. Например, когда поток SQL

воспроизводит событие на подчиненном сервере, в столбце `Info` отображается исполняемый запрос.

Если вы хотите просто поэкспериментировать с репликацией, попробуйте реализовать сценарий MySQL Sandbox, написанный Джузеппе Максиа (Giuseppe Maxia) (http://mysqlsandbox.net). Он может быстро выполнить новую установку из недавно загруженного архива MySQL. Чтобы получить работающие главный и два подчиненных сервера, достаточно нескольких нажатий клавиш и 15 секунд:

```
$ ./set_replication.pl /path/to/mysql-tarball.tar.gz
```

Инициализация подчиненного сервера на основе существующего

Ранее мы предполагали, что главный и подчиненный серверы только что установлены, поэтому данные на них практически одинаковы и позиция указателя в файле двоичного журнала известна. Но на практике такого обычно не бывает. Как правило, уже существует главный сервер, который проработал какое-то время, и требуется синхронизировать с ним новый подчиненный сервер, на котором еще нет копии данных с главного.

Существует несколько способов инициализировать, или клонировать, подчиненный сервер из имеющегося: копирование данных с главного, клонирование другого подчиненного сервера и загрузка на подчиненный сервер данных из свежей резервной копии. Чтобы синхронизировать подчиненный сервер с главным, необходимы три вещи:

- мгновенный снимок данных главного сервера в некоторый момент времени;
- текущий файл журнала главного сервера и смещение от начала этого файла в точности на то время, когда был сделан мгновенный снимок. Вместе они называются *координатами файла журнала*, так как однозначно идентифицируют позицию в двоичном журнале. Узнать координаты файла журнала главного сервера вам поможет команда `SHOW MASTER STATUS`;
- файлы двоичных журналов главного сервера с момента создания мгновенного снимка до настоящего времени.

Существует несколько способов клонировать подчиненный сервер с помощью другого сервера.

- *Холодная копия*. Самый простой способ запустить подчиненный сервер состоит в том, чтобы остановить сервер, который впоследствии станет главным, и скопировать файлы с него на подчиненный сервер (об эффективных способах копирования файлов см. приложение В). После перезапуска главный сервер откроет новый двоичный журнал, и можно будет воспользоваться командой `CHANGE MASTER TO`, указав на подчиненном сервере начало файла в качестве позиции в двоичном

журнале. Недостаток такого решения очевиден: на все время копирования главный сервер должен быть остановлен.

- *Горячая копия.* Если вы используете только таблицы MyISAM, то можно применить команды `mysqlhotcopy` или `rsync` для копирования файлов с работающего сервера. Подробную информацию см. в главе 15.
- *Использование mysqldump.* Если вы используете только таблицы InnoDB, то, чтобы выгрузить данные с главного сервера в дамп, загрузить их на подчиненный и изменить координаты репликации на подчиненном сервере в соответствии с позицией в двоичном журнале главного сервера, можно задействовать такую команду:

```
$ mysqldump --single-transaction --all-databases --master-data=1--host=server1 \ |
mysql --host=server2
```

  Флаг `--single-transaction` говорит, что при выгрузке нужно читать те данные, которые существовали на момент начала транзакции. Если имеются нетранзакционные таблицы, то для получения согласованного дампа всех таблиц следует задать флаг `--lock-all-tables`.

- *Мгновенный снимок или резервная копия.* Если известны координаты в нужном двоичном журнале, то для инициализации подчиненного сервера можно воспользоваться мгновенным снимком или резервной копией (в последнем случае необходимо иметь все двоичные журналы главного сервера с момента снятия этой копии). Восстановите данные из мгновенного снимка или копии на подчиненном сервере, а затем задайте координаты репликации с помощью команды `CHANGE MASTER TO`. Дополнительную информацию об этом способе см. в главе 15. Можно использовать мгновенные снимки `LVM`, `SAN`, `EBS`, то есть любые, которые сможете сделать.
- *Применение Percona XtraBackup.* Percona XtraBackup — это инструмент для быстрого резервного копирования с открытым исходным кодом, созданный нами несколько лет назад. Он может создавать резервные копии, не блокируя работу сервера, что делает его замечательным инструментом для настройки подчиненных серверов. Вы можете создавать подчиненные серверы путем клонирования главного или существующей реплики.

  Мы подробно рассмотрим способы использования Percona XtraBackup в главе 15, а сейчас немного расскажем о его функциональности. Просто создайте резервную копию (копируя либо главный сервер, либо уже существующую реплику) и восстановите ее на целевой машине. Затем зайдите в резервную копию и определите правильную позицию для начала репликации:

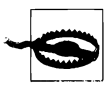
  - если вы создали резервную копию на основе главного сервера, то начинать репликацию можно с позиции, указанной в файле `xtrabackup_binlog_pos_innodb`;
  - если резервная копия выполнена на основе другого подчиненного сервера, можно начать репликацию с позиции, указанной в файле `xtrabackup_slave_info`.

  Технологии InnoDB Hot Backup и MySQL Enterprise Backup, которые рассматриваются в главе 15, — еще один удобный способ инициализировать подчиненный сервер.

- *На основе другого подчиненного сервера.* Любой из вышеупомянутых методов годится для клонирования одного подчиненного сервера из другого. Однако в этом случае флаг `--master-data` в команде `mysqldump` работать не будет.

  Еще отметим: вместо того чтобы получать координаты репликации в двоичном журнале главного сервера с помощью команды `SHOW MASTER STATUS`, следует воспользоваться командой `SHOW SLAVE STATUS` для отыскания позиции в двоичном журнале главного сервера, на которой подчиненный сервер остановился в момент снятия мгновенного снимка.

  Серьезный недостаток клонирования другого подчиненного сервера состоит в том, что если подчиненный сервер рассинхронизирован с главным, то вы будете клонировать неактуальные данные.



Не пользуйтесь командами LOAD DATA FROM MASTER и LOAD TABLE FROM MASTER! Они устарели, работают медленно и крайне опасны. К тому же применимы только к таблицам типа MyISAM.

На каком бы методе вы ни остановились, потратьте время на то, чтобы освоиться с ним, и документируйте свои действия или напишите сценарий. Не исключено, что эту процедуру придется проделать не раз, и вы не должны растеряться, если что-то пойдет не так.

Рекомендуемая конфигурация репликации

Параметров репликации много, и большинство из них так или иначе влияет на безопасность данных и производительность. Позже мы расскажем о том, какие правила можно нарушать и когда. А в этом разделе приведем рекомендуемую безопасную конфигурацию, которая сводит к минимуму шансы нарваться на неприятность.

На главном сервере самым важным параметром для двоичного журналирования является `sync_binlog`:

```
sync_binlog=1
```

При таком значении MySQL сбрасывает двоичный журнал на диск в момент коммита транзакции, поэтому в случае сбоя события журнала не потеряются. Если отключить этот режим, то работы у сервера станет немногим меньше, но при возникновении сбоя записи в журнале могут оказаться поврежденными или их вообще не будет. На подчиненном сервере, который не выступает в роли главного, этот режим приводит к излишним накладным затратам. Он применяется только к двоичному журналу, а не к журналу ретрансляции.

Если повреждение таблиц после сбоя неприемлемо, то мы рекомендуем работать с InnoDB. Подсистема хранения MyISAM хороша, если с поврежденной таблицей можно примириться, но имейте в виду, что после сбоя подчиненного сервера таблицы типа MyISAM, скорее всего, окажутся несогласованными. Есть вероятность, что

некая команда будет применена к одной или нескольким таблицам не полностью, поэтому данные останутся несогласованными даже после исправления таблиц.

Мы настоятельно рекомендуем при использовании InnoDB задавать на главном сервере следующие параметры:

```
innodb_flush_log_at_trx_commit   # Сброс после каждой записи в журнал
innodb_support_xa=1              # Только в версии MySQL 5.C и более поздних
innodb_safe_binlog               # только в версии MySQL 4.1, примерный
                                 # эквивалент innodb_support_xa
```

Это значения по умолчанию для версии MySQL 5.0 и более поздних. Также рекомендуем явно указывать базовое имя двоичного журнала, чтобы создать одинаковые имена двоичных журналов на всех серверах и предотвратить изменения имен двоичных журналов, если имя хоста сервера изменяется. Возможно, вы думаете, что нет проблемы в том, что двоичные журналы автоматически называются так же, как и сервер, но наш опыт показывает, что это вызывает большие проблемы при перемещении данных между серверами, клонировании новых реплик, восстановлении резервных копий, а также в других неожиданных случаях. Чтобы избежать этого, укажите аргумент параметра `log_bin` по желанию с абсолютным путем, но с базовым именем (как показано ранее в этой главе):

```
log_bin=/var/lib/mysql/mysql-bin  # Правильно; определили путь и базовое имя
#log_bin                          # Неправильно; базовое имя совпадет
                                  # с именем сервера
```

Мы рекомендуем включить на подчиненном сервере следующие параметры:

```
relay_log=/path/to/logs/relay-bin
skip_slave_start
read_only
```

Кроме того, советуем использовать абсолютный путь для указания местонахождения журнала ретрансляции.

Параметр `relay_log` запрещает называть файлы журналов ретрансляции так же, как хост, что позволяет избежать упомянутых ранее проблем, которые могут произойти на главном сервере, а предоставление абсолютного пути к журналам позволяет избежать ошибок в различных версиях MySQL, которые могут привести к тому, что журналы ретрансляции будут создаваться в неожиданных местах. Параметр `skip_slave_start` предотвращает автоматический перезапуск подчиненного сервера после сбоя, что оставляет возможность восстановить сервер при наличии проблем. Если же подчиненный сервер перезапускается автоматически после некорректного завершения, а база данных находится в несогласованном состоянии, то повреждение может разрастись до такой степени, что все данные придется выбросить и начать все сначала.

Параметр `read_only` не дает большинству пользователей изменять какие-либо таблицы, кроме временных. Исключение составляют поток SQL и потоки, работающие с привилегией `SUPER`. Это единственная причина, по которой обычной учетной записи имеет смысл давать привилегию `SUPER`.

Даже если вы установили все рекомендованные нами параметры, подчиненный сервер все равно может легко выйти из строя после сбоя, поскольку журналы ретрансляции и файл `master.info` не защищены от сбоев. По умолчанию они даже не сбрасываются на диск, и, более того, до версии MySQL 5.5 не существовало конфигурационного параметра, который бы это контролировал. Если используете MySQL 5.5 и вас не беспокоят дополнительные накладные затраты на дополнительные вызовы `fsync()`, стоит включить следующие параметры:

```
sync_master_info = 1
sync_relay_log = 1
sync_relay_log_info = 1
```

Если подчиненный сервер очень сильно отстает от главного, то поток ввода/вывода может создать множество журналов ретрансляции. Поток SQL удаляет их сразу после воспроизведения (это поведение можно изменить с помощью параметра `relay_log_purge`), но если отставание велико, то поток ввода/вывода вполне может заполнить весь диск. Справиться с этой проблемой поможет конфигурационный параметр `relay_log_space_limit`. Если совокупный размер всех журналов ретрансляции больше значения этого параметра, то поток ввода/вывода приостанавливается и ждет, пока поток SQL освободит место на диске.

На первый взгляд все хорошо, но здесь таится одна проблема. Если подчиненный сервер не скопировал в журнал ретрансляции все события с главного сервера, то в случае сбоя последнего они могут быть навсегда потеряны. Этот параметр в прошлом был реализован с ошибками, а кроме того, необычно используется, поэтому, если он применяется, риск ошибок возрастает. Если места на диске достаточно, то лучше дать подчиненному серверу возможность создавать журналы ретрансляции без ограничений. Именно поэтому мы не включили параметр `relay_log_space_limit` в рекомендуемую конфигурацию.

Взгляд на репликацию изнутри

Теперь, когда мы познакомились с основами репликации, можно копнуть поглубже. Рассмотрим, как на самом деле работает механизм репликации, познакомимся с его сильными и слабыми сторонами и изучим некоторые дополнительные конфигурационные параметры.

Покомандная репликация

Версии MySQL 5.0 и более ранние поддерживали только *покомандную репликацию* (она также называется *логической*). В мире баз данных это необычно. Принцип ее работы заключается в том, что протоколируются все выполненные главным сервером команды изменения данных. Когда подчиненный сервер читает из своего журнала ретрансляции событие и воспроизводит его, на самом деле он отрабатывает в точности ту же команду, которая ранее была выполнена на главном сервере. У такого решения есть свои плюсы и минусы.

Очевидный плюс — относительная легкость реализации. Простое журналирование и воспроизведение всех предложений, изменяющих данные, теоретически поддерживает синхронизацию подчиненного сервера с главным. Еще одно достоинство покомандной репликации состоит в том, что события в двоичном журнале представлены компактно. Иначе говоря, покомандная репликация задействует не слишком большую часть пропускной способности сети — запрос, обновляющий гигабайты данных, занимает всего несколько десятков байт в двоичном журнале.

На практике, однако, покомандная репликация не так проста, как кажется, поскольку многие изменения на главном сервере могут зависеть от факторов, не выраженных в тексте запроса. Например, моменты выполнения команд на главном и подчиненном серверах могут слегка — или даже сильно — различаться. Поэтому в двоичном журнале MySQL присутствует не только текст запроса, но и кое-какие метаданные, такие как временная метка. Но все равно невозможно корректно реплицировать некоторые команды, в частности запросы, в которых встречается функция `CURRENT_USER()`. Проблемы возникают также с хранимыми процедурами и триггерами.

С покомандной репликацией связана еще одна неприятность — модификации должны быть сериализуемыми. Для этого требуется больше блокировок, иногда намного больше. Не все подсистемы хранения корректно поддерживают покомандную репликацию, хотя подсистемы, включенные в официальный дистрибутив MySQL вплоть до версии 5.5, с этим справляются.

Полный перечень недостатков покомандной репликации можно найти в соответствующей главе руководства по MySQL.

Построчная репликация

В версию MySQL 5.1 была добавлена поддержка *построчной репликации*, при которой в двоичный журнал записываются фактические изменения данных, как делается в большинстве других баз данных. У такой схемы есть свои плюсы и минусы. Самое существенное достоинство заключается в том, что теперь MySQL может корректно реплицировать любую команду, причем в некоторых случаях это происходит гораздо более эффективно.

Построчное журналирование не является обратно совместимым. Утилита mysqlbinlog, входящая в дистрибутив MySQL 5.1, может читать двоичные журналы, содержащие события, журналированные в построчном формате (он не предназначен для чтения человеком, но MySQL его интерпретирует). Однако версии mysqlbinlog из предыдущих версий MySQL такой журнал не распознают и при попытке прочитать его завершаются с ошибкой.

Некоторые изменения, хранящиеся в формате построчной репликации, MySQL воспроизводит более эффективно, так как ей не приходится повторять запросы, выполненные на главном сервере. А ведь воспроизведение определенных запросов весьма

затратно. Например, следующий запрос агрегирует данные из большой таблицы, помещая результаты в таблицу меньшего размера:

```
mysql> INSERT INTO summary_table(col1, col2, sum_col3)
    -> SELECT col1, col2, sum(col3)
    -> FROM enormous_table
    -> GROUP BY col1, col2;
```

Предположим, что в таблице `enormous_table` всего три уникальные комбинации столбцов `col1` и `col2`. В этом случае запрос просматривает очень много строк в исходной таблице, но результатом является вставка лишь трех строк в конечную таблицу. Если это событие реплицировать как команду, то подчиненный сервер, чтобы вычислить эти три строки, должен будет повторить всю работу, тогда как построчная репликация обходится до смешного дешево. В данном случае построчная репликация намного эффективнее.

А следующее событие с гораздо меньшими затратами обрабатывается методом покомандной репликации:

```
mysql> UPDATE enormous_table SET col1 = 0;
```

Применение построчной репликации для данного запроса очень затратно, так как изменяются все строки, следовательно, каждую строку нужно будет записать в двоичный журнал, который вырастет до неимоверных размеров. В результате нагрузка на главный сервер резко возрастет как на этапе журналирования, так и на этапе репликации, а из-за медленной записи в журнал может пострадать конкурентность.

Поскольку ни тот, ни другой формат не идеален, MySQL 5.1 динамически переключается с одного на другой по мере необходимости. По умолчанию применяется покомандная репликация, но если обнаруживается событие, которое невозможно корректно реплицировать командой, то сервер переходит на построчную репликацию. Разрешается также явно управлять форматом с помощью сеансовой переменной `binlog_format`.

Если двоичный журнал представлен в формате построчной репликации, то восстановить данные на конкретный момент времени в прошлом становится более сложно, но все-таки возможно. В этом может помочь сервер журнала, но подробнее об этом мы поговорим в дальнейшем.

Какая репликация лучше, покомандная или построчная

Мы упомянули о преимуществах и недостатках обоих форматов репликации. Какой же из них лучше с точки зрения практического применения?

Теоретически построчная репликация, вероятно, в целом лучше, и на практике в большинстве случаев она работает отлично. Однако она реализована сравнительно недавно и, можно сказать, еще не прошла проверку временем, ее поведение в особых случаях еще не изучено должным образом, и непонятно, удовлетворяет ли она всем требованиям администраторов MySQL. Поэтому построчную репликацию пока

не выбирают в первую очередь все и каждый. Приведем более полное описание преимуществ и недостатков обоих форматов, чтобы вы могли решить, какой из них лучше подходит для ваших нужд.

- *Преимущества покомандной репликации.* Логическая репликация работает в большинстве случаев реализации на главном и подчиненном серверах разных схем. Например, как правило, ее можно заставить работать, если в таблицах содержатся разные, но совместимые типы данных, разный порядок столбцов и т. д. Это упрощает изменение схемы на подчиненном сервере, а затем копирование ее на главный, сокращая время простоев. Покомандная репликация обычно обеспечивает большую гибкость работы.

  По большому счету, процесс применения покомандной репликации является обычным выполнением SQL-команд. Это означает, что все изменения на сервере реализуются посредством хорошо понятного механизма и если что-то не работает должным образом, проверить и определить, что происходит, как правило, несложно.

- *Недостатки покомандной репликации.* Список вещей, которые не могут быть правильно реплицированы с помощью покомандного журналирования, настолько велик, что любой сервер, скорее всего, столкнется по крайней мере с одной из них. В частности, в сериях сервера 5.0 и 5.1 было много ошибок, влияющих на репликацию хранимых процедур, триггеров и т. п., — так много, что способ их репликации фактически несколько раз меняли, пытаясь заставить ее лучше работать. Подытожим: если вы используете триггеры или хранимые процедуры, не применяйте покомандную репликацию, если только не следите, как ястреб, за тем, чтобы не столкнуться с проблемами.

  Кроме того, существует множество проблем с временными таблицами, смесями подсистем хранения, конкретными конструкциями SQL, недетерминированными командами и пр. Эти проблемы варьируются от неприятных до критических.

- *Преимущества построчной репликации.* Ситуации, при которых не действует построчная репликация, встречаются значительно реже. Она правильно работает со всеми конструкциями SQL, с триггерами, хранимыми процедурами и т. д. Как правило, проблемы возникают только тогда, когда вы пытаетесь сделать что-то нетривиальное, например изменить схемы на подчиненном сервере.

  Кроме того, построчная репликация создает возможности для уменьшения блокировок, поскольку для того, чтобы ее можно было повторить, не требуется такой сильной сериализации.

  Построчная репликация работает путем регистрации измененных данных, поэтому в двоичный журнал записывается то, что фактически изменилось на главном сервере. Вам не нужно смотреть на команды и угадывать, изменили ли они какие-либо данные. Таким образом, в некотором смысле вы, как правило, знаете больше о том, что изменилось на вашем сервере, и у вас есть лучший отчет об изменениях. Кроме того, в некоторых случаях построчные двоичные журналы записывают, какими были данные, и поэтому потенциально могут оказаться полезнее при восстановлении данных.

Зачастую построчная репликация слабее нагружает процессор, так как ей не нужно планировать и выполнять запросы тем же способом (что делает покомандная репликация).

Наконец, построчная репликация иногда может помочь быстрее найти и устранить несогласованность данных. Например, покомандная репликация не выдаст ошибку, если вы обновите строку на главном сервере, но ее не будет на подчиненном. Построчная репликация в этом случае выдаст ошибку и остановится.

- *Недостатки построчной репликации.* Команды не включаются в журнал событий, поэтому будет сложно определить, какая SQL-команда выполнялась. Во многих случаях знать это так же важно, как и знать об изменении строк. (Вероятно, это будет исправлено в будущей версии MySQL.)

  Репликация изменений в подчиненных серверах выполняется совершенно по-другому — не так, как это делает SQL-команда. Фактически процесс применения изменений при построчной репликации в значительной степени является черным ящиком — не видно, что делает сервер. Кроме того, он плохо документирован и объяснен, поэтому, когда что-то работает неправильно, устранить неполадки довольно сложно. Например, если подчиненный сервер выберет неэффективный способ поиска строк для изменения, вы этого не заметите.

  Если у вас есть несколько уровней серверов репликации и все они настроены на построчное журналирование, команда, которую вы выполняете, когда переменная `@@binlog_format` на уровне сеанса установлена в `STATEMENT`, будет журналироваться как команда на сервере, на котором она возникает, но подчиненные серверы первого уровня могут передать событие в построчном формате для дальнейших реплик в цепочке. То есть желаемое покомандное журналирование после прохождения через топологию репликации вновь будет переключено на построчное.

  Построчное журналирование, в отличие от покомандного, не может обрабатывать некоторые процессы, например изменения схемы в подчиненных серверах.

  Построчная репликация иногда останавливается в тех случаях, когда покомандная репликация будет продолжаться, например, когда на подчиненном сервере отсутствует строка, которую требуется изменить. Это можно было бы считать достоинством. В любом случае такое поведение настраивается с помощью параметра `slave_exec_mode`.

  Многие из этих недостатков с течением времени будут исправлены, но на момент написания книги они все еще присутствуют в большинстве промышленных систем.

Файлы репликации

Теперь рассмотрим некоторые файлы, участвующие в процессе репликации. О двоичном журнале и журнале ретрансляции вы уже знаете, но есть и другие файловые объекты. Конкретное место их хранения зависит в основном от конфигурационных параметров MySQL. В разных версиях MySQL каталоги по умолчанию

различны. Вы, скорее всего, найдете их либо в каталоге данных, либо в каталоге, где находится `.pid`-файл сервера (в UNIX-системах это обычно каталог `/var/run/mysqld/`). Перечислим их.

- `mysql-bin.index`. Если запись в двоичный журнал включена, то сервер создает файл с таким же именем, как у двоичных журналов, но с расширением `.index`. В нем регистрируются все файлы двоичных журналов, имеющиеся на диске. Это не индекс в том смысле, в каком мы говорим об индексах таблиц, — он просто состоит из текстовых строк, в каждой из которых указано имя одного файла двоичного журнала.

  Вероятно, у вас возникла мысль, что этот файл лишний и может быть удален (в конце концов, MySQL может просто найти все файлы на диске). Не делайте этого! MySQL игнорирует двоичные журналы, не указанные в индексном файле.
- `mysql-relay-bin.index`. Этот файл играет ту же роль для журналов ретрансляции, что и рассмотренный ранее файл — для двоичных журналов.
- `master.info`. В этом файле хранится информация, необходимая подчиненному серверу для соединения с главным. Формат текстовый (по одному значению в строке), и он различен в разных версиях MySQL. Не удаляйте его, иначе после перезапуска подчиненный сервер не будет знать, как подключиться к главному. Поскольку в этом файле может храниться пароль пользователя в открытом виде, разумно будет максимально ограничить права доступа к нему.
- `relay-log.info`. В этом файле на подчиненном сервере хранятся имя его текущего двоичного журнала и координаты репликации (то есть место в двоичном журнале главного сервера, до которого дошел подчиненный). Не удаляйте этот файл, иначе после перезапуска подчиненный сервер не будет знать, с какого места продолжить репликацию, и попытается воспроизвести уже выполненные команды.

Совокупность этих файлов представляет собой довольно прямолинейный способ сохранить состояние репликации и журналов. К сожалению, запись в них производится несинхронно, поэтому, если произойдет сбой питания в момент, когда файлы не были сброшены на диск, то после перезапуска данные в них окажутся некорректными. Как мы уже говорили, это улучшено в MySQL 5.5.

На `.index`-файлы влияет также параметр `expire_logs_days`, определяющий, сколько времени MySQL должен сохранять уже закрытые двоичные журналы. Если в файле `mysql-bin.index` упоминаются файлы, отсутствующие на диске, то автоматическое удаление в некоторых версиях MySQL работать не будет, даже команда `PURGE MASTER LOGS` ничего не даст. В общем случае для решения этой проблемы нужно поручить управление двоичными журналами самому серверу MySQL, уж он-то не запутается. (То есть нельзя использовать утилиту rm для удаления файлов.)

Необходимо выработать стратегию удаления старых журналов — с помощью параметра `expire_logs_days` или иными средствами, — иначе рано или поздно MySQL заполнит двоичными журналами весь диск. При этом следует учитывать и принятую в организации политику резервного копирования.

Отправка событий репликации другим подчиненным серверам

Параметр `log_slave_updates` позволяет использовать подчиненный сервер в роли главного для других подчиненных. Он заставляет сервер MySQL записывать события, выполняемые потоком SQL, в собственный двоичный журнал, доступный подчиненным ему серверам. Эта схема изображена на рис. 10.2.

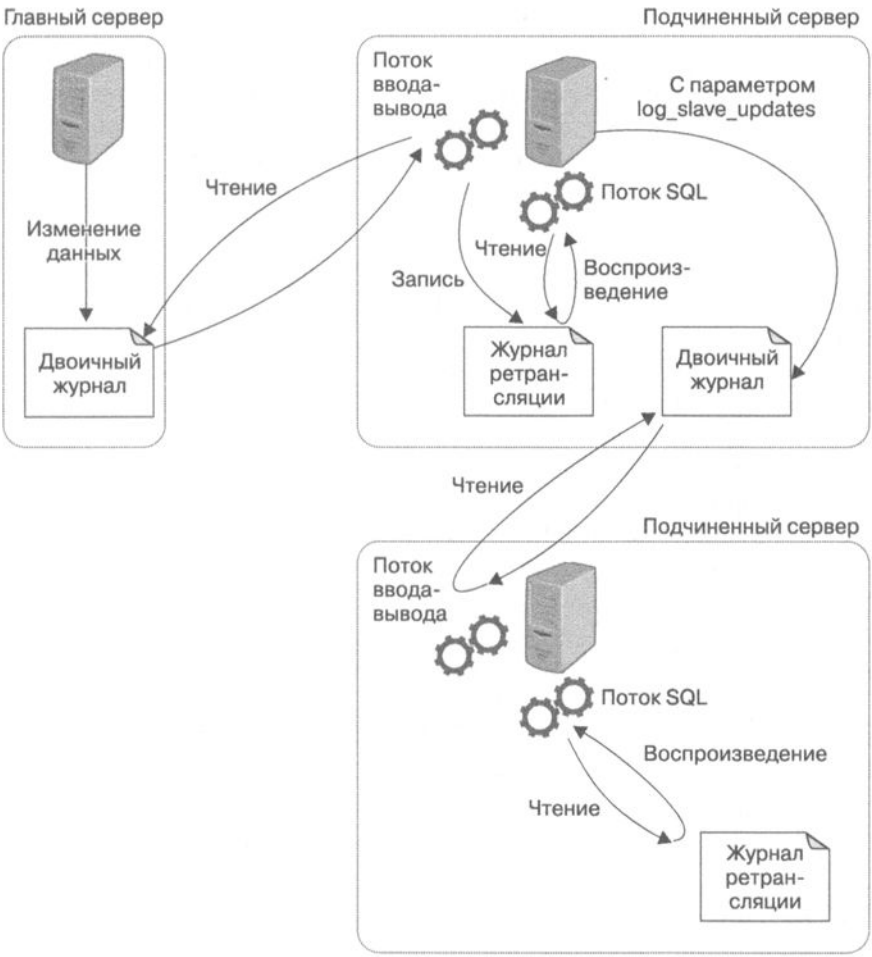


Рис. 10.2. Передача события репликации по цепочке подчиненных серверов

В данном случае любое изменение на главном сервере приводит к записи события в его двоичный журнал. Первый подчиненный сервер извлекает и исполняет это событие. На этом жизнь события и завершилась бы, но, поскольку включен режим `log_slave_updates`, подчиненный сервер записывает его в свой двоичный журнал.

Теперь второй подчиненный сервер может извлечь это событие и поместить в свой журнал ретрансляции. Такая конфигурация означает, что изменения, выполненные на главном сервере, распространяются по цепочке подчиненных серверов, не подключенных напрямую к главному. Мы предпочитаем по умолчанию оставлять режим `log_slave_updates` включенным, так как это позволяет подключать подчиненный сервер без перезапуска сервера.

Когда первый подчиненный сервер переписывает событие из двоичного журнала главного сервера в свой двоичный журнал, его позиция в журнале почти наверняка изменится, то есть она может либо оказаться в другом файле, либо еще как-то иначе сместиться. Поэтому нельзя предполагать, что все серверы, логически находящиеся в одной и той же точке репликации, имеют одинаковые координаты репликации. Позже мы увидим, что это существенно осложняет решение некоторых задач, например подключение подчиненного сервера к другому главному или преобразование подчиненного сервера в главный.

Если не позаботиться о том, чтобы у каждого сервера был уникальный идентификатор, то подобное конфигурирование подчиненного сервера может стать причиной трудноуловимых ошибок и даже привести к полной остановке репликации. Часто спрашивают: зачем нужно присваивать серверу идентификатор? Разве MySQL не может реплицировать команды, не зная их происхождения? Почему MySQL хочет, чтобы идентификатор сервера был глобально уникальным? Дело в том, что нужно предотвратить бесконечные циклы в процедуре репликации. Когда поток SQL на подчиненном сервере читает журнал ретрансляции, он отбрасывает все события, в которых идентификатор сервера совпадает с его собственным. Тем самым бесконечный цикл разрывается. Предотвращение бесконечных циклов важно в таких топологиях репликации, как «главный сервер — главный сервер»[1].

При возникновении затруднений с настройкой репликации обратите внимание на идентификатор сервера. Недостаточно просто проверить переменную @@server_id. У нее всегда есть какое-то значение по умолчанию, но репликация не будет работать, если значение не задано явно в файле my.cnf или командой SET. Если вы используете команду SET, не забудьте также обновить конфигурационный файл, иначе измененные настройки пропадут после перезапуска сервера.

Фильтры репликации

Параметры фильтрации позволяют реплицировать только часть данных, хранящихся на сервере, что, возможно, не оправдывает ваших ожиданий. Есть два вида фильтров репликации: одни применяются при записи событий в двоичный журнал на главном

[1] Команды, выполняемые в бесконечных циклах, также являются одной из многих радостей топологий многосерверной кольцевой репликации, которые мы покажем в дальнейшем. Бегите от кольцевой репликации как от чумы.

сервере, другие — при чтении событий из журнала ретрансляции на подчиненном сервере. И те и другие изображены на рис. 10.3

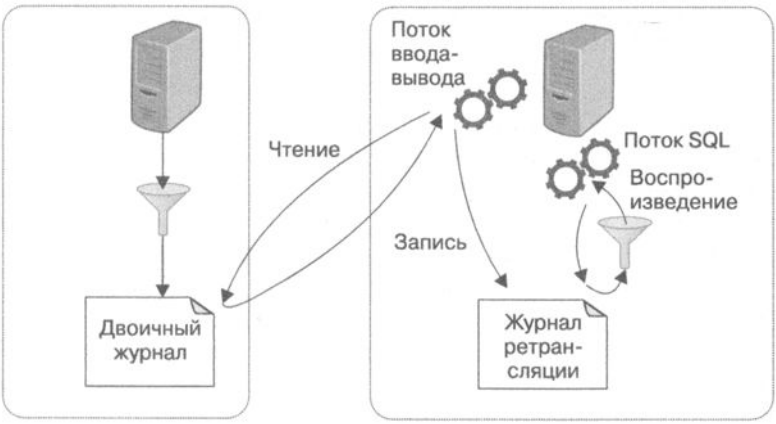


Рис. 10.3. Параметры фильтров репликации

Фильтрацией двоичного журнала управляют параметры `birlog_do_db` и `binlog_ignore_db`. Но, как мы скоро поясним, их не надо включать, если конечно вы не хотите постоянно объяснять шефу, почему данные пропадают и их нельзя восстановить.

На подчиненном сервере параметры `replicate_*` управляют фильтрацией событий, считываемых потоком SQL из журнала ретрансляции. Можно реплицировать или игнорировать одну или несколько баз данных, переписывать одну базу данных в другую и реплицировать или игнорировать определенные таблицы, задаваемые с помощью паттерна `LIKE`.

Очень важно понимать, что параметры `*_do_db` и `*_ignore_db` как на главном, так и на подчиненном сервере работают не так, как можно было бы ожидать. Естественно полагать, что фильтрация производится по имени базы данных объекта, но на самом деле анализируется имя текущей базы данных по умолчанию[1]. Иными словами, если выполнить на главном сервере такие команды:

```
mysql> USE test;
mysql> DELETE FROM sakila.film;
```

то параметры `*_do_db` и `*_ignore_db` будут отфильтровывать команду `DELETE` в базе `test`, а не в базе `sakila`. Обычно это совсем не то, что требуется, а в результате будут реплицироваться или игнорироваться не те команды. Хотя у параметров `*_do_db` и `*_ignore_db` есть полезное применение, требуются они редко, так что будьте осторожны. Если вы ими воспользуетесь, то реплики очень легко могут оказаться рассогласованными или начать сбоить.

[1] Если вы используете покомандную репликацию. Если же применяете построчную репликацию, учтите, что они ведут себя по-разному (еще одна причина держаться от них подальше).

Параметры binlog_do_db и binlog_ignore_db не только могут нарушить репликацию, но и делают невозможным восстановление данных на конкретный момент времени в прошлом. В большинстве ситуаций не применяйте их. Они могут вызвать бесконечные проблемы. Далее в этой главе мы покажем несколько альтернативных способов фильтрации событий репликации с помощью таблиц Blackhole.

В целом, фильтры репликации — это бомба замедленного действия. Предположим, вы хотите, чтобы изменения привилегий не распространялись на реплики. Это вполне нормальная цель (такое желание может натолкнуть вас на мысль, что вы делаете что-то не так — возможно, есть и другие способы достичь вашей настоящей цели). Конечно, команды `GRANT` вы подобным образом отфильтруете, но заодно не будут реплицироваться события и подпрограммы. Вот из-за таких непредвиденных последствий мы и призываем быть очень аккуратными, работая с фильтрами. Возможно, стоит вместо этого воспрепятствовать репликации конкретных команд, обычно с помощью команды `SET SQL_LOG_BIN=0`, но с такой практикой сопряжены другие опасности. В общем случае к фильтрам репликации следует подходить с особой осторожностью и применять их только в случае острой необходимости, потому что при этом легко нарушить репликацию и вызвать проблемы, которые проявятся в самый неподходящий момент, например во время аварийного восстановления.

Параметры фильтрации хорошо документированы в руководстве по MySQL, поэтому не станем здесь повторяться.

Топологии репликации

MySQL позволяет настроить репликацию чуть ли не для любой конфигурации главных и подчиненных серверов с одним ограничением: у каждого подчиненного сервера может быть только один главный. Возможны различные сложные топологии, но даже совсем простые обладают немалой гибкостью. Одну и ту же топологию можно применять множеством способов. Их разнообразия хватило бы на целую книгу.

Вы уже видели, как настроить главный сервер с одним подчиненным. В этом разделе рассмотрим другие распространенные топологии и обсудим их сильные стороны и ограничения. По ходу изложения не забывайте о нескольких базовых правилах.

- У каждого подчиненного сервера MySQL может быть только один главный.
- У каждого подчиненного сервера должен быть уникальный идентификатор.
- Один главный сервер может иметь много подчиненных (иными словами, у подчиненного сервера может быть много родственников).
- Подчиненный сервер может распространять полученные от главного изменения далее, то есть выступать в роли главного сервера для своих подчиненных. Для этого следует включить режим `log_slave_updates`.

Один главный сервер с несколькими подчиненными

Если исключить рассмотренную ранее конфигурацию с двумя серверами «главный — подчиненный», то эта топология самая простая. На самом деле она ничуть не сложнее базовой: подчиненные серверы никак не взаимодействуют между собой[1], все подчиненные серверы подключены к главному. Такая схема изображена на рис. 10.4.

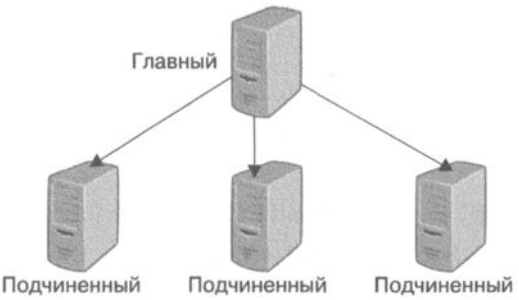


Рис. 10.4. Один главный сервер с несколькими подчиненными

Эта конфигурация наиболее полезна, когда операций записи мало, а операций чтения много. Вы можете распределить операции чтения среди любого количества подчиненных серверов вплоть до того момента, когда реплики слишком сильно нагрузят главный сервер или возникнут проблемы с пропускной способности сети от главного сервера до реплик. Подчиненные серверы можно включить все сразу или добавлять постепенно, как описано ранее в этой главе.

Хотя указанная топология очень проста, ее гибкости достаточно для решения различных задач. Приведем несколько идей.

- Задействуйте подчиненные серверы для разных ролей (например, можно построить другие индексы или использовать другие подсистемы хранения).
- Настройте один из подчиненных серверов в качестве резервного на случай выхода главного из строя. Никаких других действий, кроме репликации, на нем производиться не будет.
- Поставьте один подчиненный сервер в удаленный центр обработки данных на случай аварийного восстановления.
- Настройте задержку применения изменений на подчиненном сервере для аварийного восстановления данных.
- Используйте один из подчиненных серверов для резервного копирования, обучения, разработки или предварительной подготовки данных

Одна из причин популярности данной топологии — то, что она позволяет избежать сложностей, присущих другим конфигурациям. Например, легко сравнить один подчиненный сервер с другим, исходя из позиций в двоичном журнале на главном

[1] Строго говоря, это не так. Если у них одинаковые идентификаторы, они будут драться между собой и периодически выпихивать друг друга с главного сервера.

сервере, поскольку все они должны быть одинаковыми. Иными словами, если остановить все подчиненные серверы в одной и той же логической точке репликации, то окажется, что все они читают из одного физического места в журнале главного сервера. Это весьма полезное свойство упрощает ряд административных задач, в частности преобразование подчиненного сервера в главный.

Упомянутым свойством характеризуются только подчиненные родственники. Сравнивать позиции в журнале для серверов, не связанных прямым отношением «главный — подчиненный» и не являющихся родственниками, гораздо сложнее. Во многих рассматриваемых далее топологиях, например в древовидной репликации или конфигурации с главным сервером-распределителем, трудно понять, в какой точке логической последовательности событий находится подчиненный сервер.

«Главный сервер — главный сервер» в режиме «активный — активный»

Репликация типа «главный — главный» (или репликация с двумя главными, или двунаправленная репликация) подразумевает наличие двух серверов, каждый из которых сконфигурирован одновременно и как главный, и как подчиненный по отношению к другому. Другими словами, это пара соруководителей. На рис. 10.5 показана настройка.

Рис. 10.5. Репликация «главный сервер — главный сервер»

Многоресурсная репликация

Мы используем термин «репликация с несколькими главными серверами» для описания ситуации, когда один подчиненный сервер связан с несколькими главными. Что бы вам ни говорили по этому поводу, MySQL, в отличие от некоторых других серверов баз данных, пока не поддерживает конфигурацию, изображенную на рис. 10.6. Однако далее мы все же покажем, как можно эмулировать такую конфигурацию.

Рис. 10.6. MySQL не поддерживает репликацию с несколькими главными серверами

Репликация типа «главный — главный» в режиме «активный — активный» находит применение, но обычно узкоспециализированное. Например, ее можно использовать в географически разнесенных отделениях, в каждом из которых необходимо изменять данные.

Основная проблема, которая возникает в этом случае, — как обрабатывать конфликтующие изменения. Но перечень проблем, связанных с наличием двух равноправных главных серверов, этим далеко не исчерпывается. Обычно сложности появляются, когда одна и та же строка одновременно изменяется на обоих серверах или в один и тот же момент времени выполняется вставка в таблицу с автоинкрементным столбцом[1].

В версии MySQL 5.0 были добавлены средства, сделавшие этот вид репликации немного безопаснее: параметры `auto_increment_increment` и `auto_increment_offset`. Они позволяют серверам автоматически генерировать неконфликтующие значения в запросах `INSERT`. Но все равно разрешать запись на обоих главных серверах опасно. Если на двух машинах обновления производятся в разном порядке, то данные могут незаметно рассинхронизироваться. Пусть, например, имеется таблица с одной строкой и одним столбцом, содержащая значение `1`. Предположим, что одновременно выполняются две команды:

- на первом главном:

```
mysql> UPDATE tbl SET col=col + 1;
```

- на втором главном:

```
mysql> UPDATE tbl SET col=col * 2;
```

Что мы получим? На одном сервере значение `4`, на другом — `3`. А главное, ни о каких ошибках подсистема репликации не сообщит.

Рассинхронизация данных — это еще цветочки. Что, если репликация по какой-то причине остановится, а приложения на обоих серверах продолжат обновлять данные? В этом случае ни один из них нельзя взять в качестве основы для клонирования при синхронизации, так как на каждом есть изменения, которых нет на другом. Выпутаться из такой ситуации крайне сложно. Будьте осторожны!

Отчасти эти проблемы можно обойти, если тщательно продумать схему секционирования данных и распределения привилегий и применять ее только в том случае, когда вы действительно знаете, что делаете[2]. Но это трудно, и обычно существуют другие способы достичь желаемого результата.

В общем случае разрешение записи на обоих серверах создает больше проблем, чем решает. Однако в режиме «активный — пассивный» эта конфигурация может быть весьма полезной, что мы и продемонстрируем в следующем разделе.

---

[1] На самом деле эти проблемы обычно возникают в 3 часа ночи в выходные, а их решение занимает месяцы.

[2] Но только отчасти — мы можем взять на себя роль адвоката дьявола и указать на изъяны почти в любой мыслимой конфигурации.

«Главный сервер — главный сервер» в режиме «активный — пассивный»

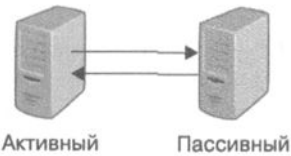


Рис. 10.7. Репликация «главный сервер — главный сервер» в режиме «активный — пассивный»

У репликации типа «главный — главный» есть вариант, который позволяет обойти все подводные камни, рассмотренные ранее. Более того, это очень мощный способ разработки отказоустойчивых и высоконадежных систем. Он заключается в том, что один из серверов является пассивным — сервером только для чтения (рис. 10.7).

Такая схема позволяет без труда менять активный и пассивный серверы ролями, поскольку их конфигурации симметричны. В свою очередь, это дает возможность без проблем аварийно переключиться на резервный и вернуться на основной сервер. Кроме того, вы можете обслуживать базу, оптимизировать таблицы, переходить на новую версию операционной системы (приложения, оборудования) и решать другие задачи, не останавливая работу.

Например, команда `ALTER TABLE` блокирует таблицу полностью, запрещая для нее чтение и запись. Это может занять много времени, в течение которого обслуживание будет прервано. Однако, имея конфигурацию с несколькими главными серверами, вы можете остановить потоки репликации на активном сервере, так что он не будет обрабатывать обновления от пассивного сервера, изменить на пассивном сервере таблицу, поменять серверы ролями и запустить потоки репликации на сервере, который раньше был активным[1]. Теперь этот сервер прочтет свой журнал ретрансляции и выполнит ту же самую команду `ALTER TABLE`. Это, конечно, займет столь же много времени, но это неважно, поскольку серверу не нужно обслуживать запросы.

Конфигурация «главный — главный» в режиме «активный — пассивный» позволяет обойти и многие другие проблемы и ограничения MySQL. Для решения такой задачи также существует несколько наборов инструментов.

Разберемся, как конфигурируется пара «главный — главный». Описанные далее действия нужно выполнить на обоих серверах, чтобы итоговые конфигурации были симметричными.

1. Убедиться, что на обоих серверах находятся совершенно одинаковые данные.
2. Включить запись в двоичный журнал, назначить серверам уникальные идентификаторы и добавить учетные записи для репликации.
3. Включить режим журналирования обновлений подчиненного сервера. Как вы вскоре увидите, это очень важно для аварийного переключения на резервный сервер и обратно.

---

[1] Можно не останавливать репликацию, а на время отключить запись в двоичный журнал командой SET SQL_LOG_BIN=0. Кроме того, некоторые команды, к примеру OPTIMIZE TABLE, поддерживают режимы LOCAL или NO_WRITE_TO_BINLOG, останавливающие запись в двоичный журнал. Тем самым вы можете точнее выбирать время работы, а не просто позволять команде ALTER выполняться тогда, когда это происходит в потоке репликации.

4. Не обязательно — сконфигурировать пассивный сервер в режиме чтения во избежание изменений, которые могли бы конфликтовать с изменениями на активном сервере.
5. Запустить MySQL на обоих серверах.
6. Сконфигурировать каждый сервер так, чтобы он был подчиненным для другого, начав с пустого двоичного журнала.

Теперь проследим за тем, что происходит, когда на активном сервере производится какое-либо изменение. Оно записывается в его двоичный журнал и реплицируется в журнал ретрансляции пассивного сервера. Пассивный сервер выполняет запрос и записывает событие в собственный двоичный журнал, поскольку режим `log_slave_updates` включен. Затем активный сервер игнорирует это событие, так как идентификатор сервера совпадает с его собственным. (Подробнее о том, как поменять роли, см. в подразделе «Смена главного сервера» далее в этой главе.)

В некотором смысле топологию «главный — главный» в режиме «активный — пассивный» можно считать способом горячего резервирования с тем, однако, отличием, что резервный сервер иногда используется для повышения производительности. С него можно читать данные, выполнять на нем резервное копирование, обслуживание в офлайновом режиме, устанавливать новые версии программного обеспечения и т. д. Ничего этого на настоящем сервере горячего резерва делать нельзя. Однако такая конфигурация не дает возможности повысить производительность записи по сравнению с одиночным сервером (мы еще скажем несколько слов по этому поводу).

По мере рассмотрения других сценариев и способов применения репликации мы будем возвращаться к этой конфигурации. Она широко распространена и очень полезна.

«Главный сервер — главный сервер с подчиненными»

Существует похожая конфигурация, при которой у каждого главного сервера есть один или несколько подчиненных (рис. 10.8).

Рис. 10.8. Топология «главный сервер — главный сервер с подчиненными»

Достоинством этой конфигурации является дополнительная избыточность. В случае репликации между территориально удаленными центрами она позволяет устранить единственную точку отказа в каждом центре. Кроме того, как обычно, на подчиненных серверах можно выполнять запросы, читающие много данных.

Но даже если вы собираетесь применять топологию «главный — главный» только для переключения на резервный сервер при отказе, эта конфигурация все равно полезна. Один из подчиненных серверов может взять на себя роль отказавшего главного, хотя реализовать это несколько труднее. Можно также переподчинить один из серверов другому главному. Впрочем, нужно принимать во внимание повышенную сложность этих действий.

Кольцевая репликация

Конфигурация с двумя главными серверами на самом деле лишь частный случай[1] кольцевой репликации (рис. 10.9). В кольце есть три главных сервера или более. Каждый сервер выступает в роли подчиненного для предшествующего ему сервера и в роли главного — для последующего. Такая топология также называется *круговой репликацией*.

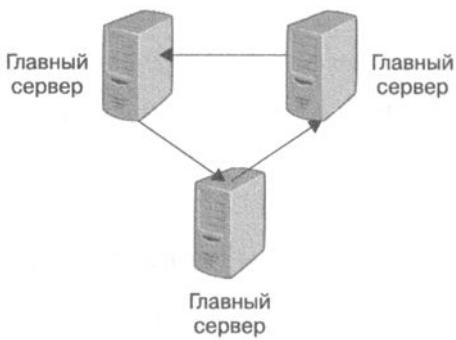


Рис. 10.9. Топология кольцевой репликации

Кольцо не обладает некоторыми существенными достоинствами топологии «главный — главный», например симметричностью конфигурации и простотой аварийного переключения. Кроме того, для работы такой системы необходимо, чтобы все входящие в кольцо серверы были доступны, а это существенно увеличивает вероятность ее отказа. Если удалить из кольца один узел, то порожденные им репликационные события могут курсировать по кольцу бесконечно, так как отфильтровать событие может лишь создавший его сервер. Таким образом, кольца по природе своей хрупки, и, несмотря на вашу ловкость, лучше к ним не прибегать.

В какой-то степени снизить риски, присущие кольцевой репликации, можно, добавив в каждом узле подчиненные серверы и тем самым увеличив избыточность

[1] Чуть более осмысленный частный случай, могли бы мы добавить.

(рис. 10.10). Но это лишь страховка на случай отказа сервера. Выключение питания или иные проблемы, приводящие к исчезновению соединения между узлами, все равно разрушают кольцо.

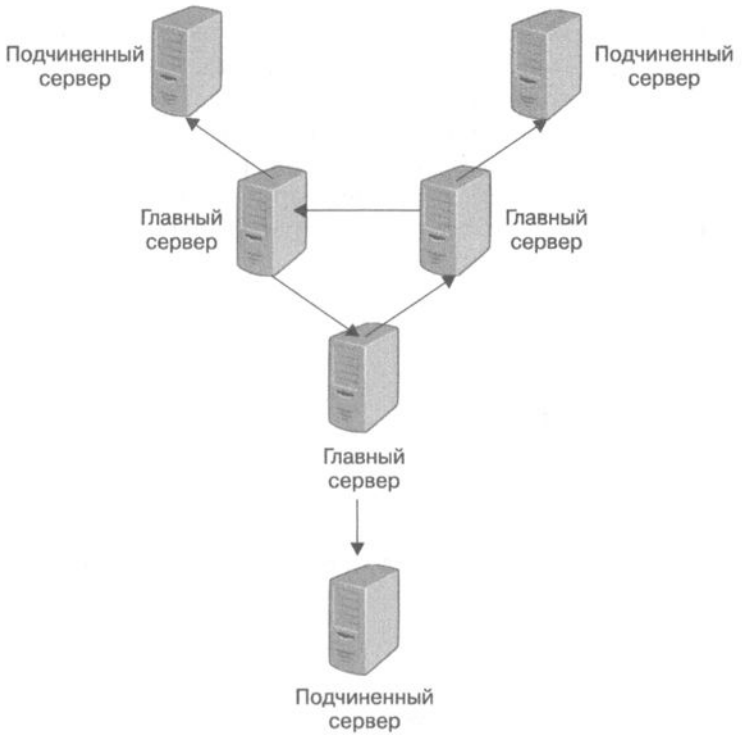


Рис. 10.10. Кольцевая репликация с дополнительными подчиненными серверами в каждом узле

Главный сервер, главный сервер-распространитель и подчиненные

Мы уже отмечали, что при наличии большого количества подчиненных серверов нагрузка на главный может оказаться чрезмерной. Каждый подчиненный сервер создает на главном отдельный поток, который выполняет специальную команду `binlog dump`. Эта команда читает данные из двоичного журнала и посылает их подчиненному серверу. Для каждого подчиненного потока работа дублируется, никакого совместного использования ресурсов, необходимых команде дампа, не предусмотрено.

Если подчиненных серверов много и в двоичном журнале встретится какое-то особо громоздкое событие, например `LOAD DATA INFILE` для очень длинного файла, то нагрузка на главный сервер может резко возрасти. У главного сервера может даже закончиться память, поскольку все подчиненные запрашивают это событие одновременно. Итог — аварийное завершение. В то же время, если все подчиненные

серверы извлекают из двоичного журнала разные события, которые уже вытеснены из кэша файловой системы, возрастает количество операций поиска на диске, что тоже может негативно повлиять на производительность главного сервера и привести к конкуренции на мьютексе.

Так что, если необходимо много подчиненных серверов, часто имеет смысл разгрузить главный сервер и ввести в схему так называемый *главный сервер-распространитель*. Это подчиненный сервер, который выполняет одну-единственную функцию — читать двоичные журналы с главного сервера и передавать их дальше. К серверу-распространителю можно подключить много подчиненных, сняв тем самым нагрузку с основного сервера. Чтобы не тратить ресурсы распространителя на выполнение запросов, нужно задать на нем для всех таблиц подсистему хранения Blackhole (рис. 10.11).

Рис. 10.11. Главный сервер, главный сервер-распространитель и много подчиненных

Трудно определенно сказать, сколько подчиненных серверов может обслужить главный без распространителя. В качестве грубого ориентира можно принять следующее соображение: если пропускная способность главного сервера используется практически полностью, то не стоит присоединять к нему больше десяти подчиненных. Если количество операций записи очень мало или реплицируется лишь несколько таблиц, то главный сервер, наверное, сможет обслужить и больше подчиненных. Кроме того, не обязательно ограничиваться только одним распространителем. Если количество подчиненных серверов очень велико, то можно организовать и несколько распространителей и даже создать из них пирамиду. В некоторых случаях снять часть нагрузки с главного сервера помогает установка параметра `slave_compressed_protocol`. Это наиболее полезно для перекрестной репликации между дата-центрами.

Серверы-распространители можно задействовать и для других целей, например задав на них правила фильтрации и перезаписи событий в двоичном журнале. Это гораздо эффективнее, чем повторять запись в журнал, перезапись и фильтрацию на каждом подчиненном сервере.

Если на сервере-распространителе используется подсистема хранения Blackhole, то количество обслуживаемых им подчиненных серверов можно увеличить. Распространитель станет выполнять запросы, но затраты на них будут минимальными, так как в таблицах типа Blackhole никакие данные не хранятся. Недостатком таблиц Blackhole является то, что у них есть ошибки. Например, в некоторых случаях автоинкрементные идентификаторы не помещаются в двоичные журналы. Поэтому при использовании таблиц Blackhole будьте очень осторожны[1].

Часто задают вопрос: как гарантировать, что все таблицы на сервере-распространителе используют подсистему хранения Blackhole? Что, если кто-то создаст на главном сервере новую таблицу, указав другую подсистему хранения? Вообще эта проблема возникает всякий раз, когда нужно сделать разными подсистемы хранения на главном и подчиненном серверах. Обычно для ее решения на сервере задают следующий параметр:

```
storage_engine = blackhole
```

Он распространяется только на команды `CREATE TABLE`, в которых подсистема хранения не указана явно. Если вы пользуетесь неподконтрольным вам приложением, то такая топология может оказаться хрупкой. Можно с помощью параметра `skip_innodb` запретить использование InnoDB, тем самым заменив подсистему хранения на MyISAM, но отключить подсистемы MyISAM и Memory не получится.

Еще один существенный недостаток — сложность подмены главного сервера одним из подчиненных (конечных). Сделать подчиненный сервер главным трудно, так как из-за наличия промежуточного сервера координаты в двоичном журнале почти наверняка будут не такими, как на исходном главном сервере[2].

Дерево или пирамида

Если главный сервер реплицируется на множество подчиненных — неважно, с целью распределения данных по географическому признаку или просто для повышения пропускной способности чтения, — может оказаться удобной пирамидальная топология (рис. 10.12).

---

[1] Для начала взгляните на ошибки 35 178 и 62 829. В целом, если вы используете нестандартную подсистему хранения или функцию, поищите, что пишут об уже исправленных и еще не исправленных в них ошибках.

[2] Для создания приближенного глобального идентификатора транзакции можно использовать утилиту pt-heartbeat из пакета Percona Toolkit. Это значительно упрощает поиск позиции в двоичном журнале на разных серверах, потому что сама таблица heartbeat содержит приблизительные позиции двоичного журнала.

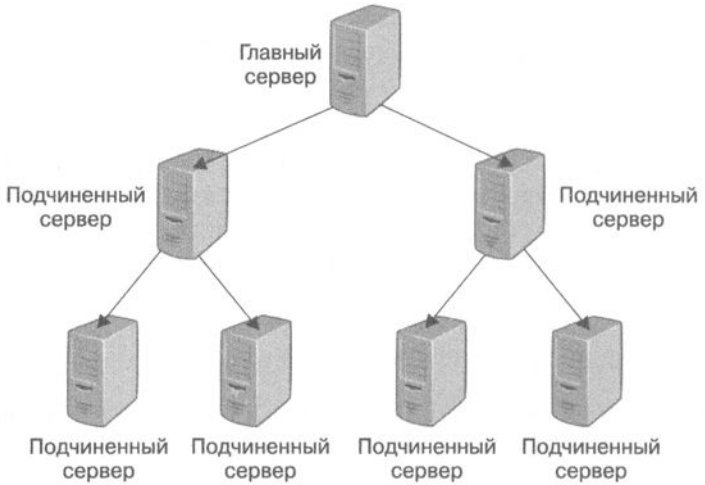


Рис. 10.12. Топология репликации «пирамида»

Преимущество такой конфигурации состоит в том, что разгружается главный сервер, как и в случае применения распределителя. Недостаток же таков: отказ на промежуточном уровне распространяется сразу на несколько серверов, чего не произошло бы, будь все они подключены к главному напрямую. К тому же чем больше промежуточных уровней, тем сложнее восстановление после сбоев.

Пользовательские схемы репликации

Механизм репликации в MySQL достаточно гибок для того, чтобы можно было создавать пользовательские схемы, приспособленные под нужды конкретного приложения. Как правило, применяется комбинация фильтрации, распространения и репликации с использованием различных подсистем хранения. Можно также прибегать к различным трюкам, например задавать для таблиц на исходном или конечном сервере подсистему хранения Blackhole (как описано ранее). Схема может быть сколь угодно изощренной. Нужно лишь стараться, чтобы сложность мониторинга и администрирования оставалась в разумных пределах, и учитывать имеющиеся ресурсы (пропускную способность сети, мощность процессора и т. д.).

Избирательная репликация

Чтобы в полной мере использовать локальность ссылок и уместить рабочее множество для запросов на чтение в памяти, можно реплицировать небольшие порции данных на множество подчиненных серверов. Если на каждом подчиненном сервере находится лишь малая часть информации, то, разослав запросы на чтение всем таким серверам, можно добиться гораздо более эффективного использования памяти на каждом подчиненном сервере. Кроме того, на каждую реплику будет приходиться лишь часть общей нагрузки, порождаемой операциями записи на главном сервере, так что главный сервер можно будет сделать более мощным, не опасаясь, что подчиненные отстанут.

Эта схема напоминает горизонтальное секционирование данных, о котором мы подробно поговорим в следующей главе, но ее достоинство в том, что все же имеется один сервер — главный, на котором хранятся все данные. Это означает, что любая операция записи производится только на одном сервере, а если для некоторой операции чтения не существует единственного подчиненного сервера, содержащего все необходимые данные, то всегда можно выполнить ее на главном сервере. Даже если все операции чтения невозможно выполнить на подчиненных серверах, то за счет переноса большей их части все-таки можно разгрузить главный.

Простейший способ реализовать эту схему заключается в том, чтобы поместить информацию в разные базы на главном сервере, а потом реплицировать каждую базу на разные подчиненные серверы. Например, можно разнести по подчиненным серверам данные, относящиеся к различным подразделениям компании: `sales`, `marketing`, `procurement` и т. д. Тогда на каждом подчиненном сервере нужно будет задать конфигурационный параметр `replicate_wild_do_table`, который ограничивает воспринимаемые им данные одной базой. Вот как это может выглядеть для базы данных sales:

```
replicate_wild_do_table = sales.%
```

Полезна также фильтрация на сервере-распространителе. Например, если требуется реплицировать какую-то часть сильно загруженного сервера по медленной или очень затратной сети, то можно организовать локальный распространитель с таблицами типа Blackhole и правилами фильтрации. Фильтры на этом сервере будут удалять ненужные записи из журналов. Это позволит одновременно избежать задания небезопасных параметров журналирования на главном сервере и передачи всех журналов по сети удаленным подчиненным.

Разграничение функций

Многие приложения представляют собой смесь оперативной обработки транзакций (OLTP) и оперативной аналитической обработки (OLAP). OLTP-запросы обычно бывают короткими и транзакционными. Напротив, OLAP-запросы большие, выполняются медленно, зато не требуют стопроцентно актуальных данных. Запросы разных видов по-разному нагружают сервер. Поэтому лучше выполнять их на различных серверах с разными конфигурациями и, возможно, даже разными подсистемами хранения и оборудованием.

Часто эту проблему решают путем репликации данных с OLTP-сервера на подчиненные серверы, специально настроенные для работы в условиях OLAP-нагрузки. На них может быть установлено другое оборудование, построены другие индексы и применены иные подсистемы хранения. Если для OLAP-запросов выделяется отдельный подчиненный сервер, то, возможно, придется смириться с более значительным отставанием репликации или каким-то другим снижением качества обслуживания на нем. Зато этот сервер можно использовать для задач, которые на невыделенном подчиненном сервере работали бы недопустимо долго, например для выполнения очень долгих запросов.

Никакой специальной схемы репликации для этого придумывать не надо, хотя можно не реплицировать часть данных с главного сервера, если это позволит добиться

ощутимой экономии на подчиненном сервере. Напомним, что игнорирование даже небольшой части данных за счет фильтров репликации на журнале ретрансляции может заметно снизить объем ввода/вывода и активность кэша.

Архивация данных

На подчиненном сервере можно хранить архив данных, удаленных с главного. Для этого нужно выполнить на главном сервере команду удаления так, чтобы она не была повторена на подчиненном. Это можно сделать двумя способами: выборочно отключить запись в двоичный журнал на главном сервере или сконфигурировать правила `replicate_ignore_db` на подчиненном. (Да, оба способа небезопасны.)

В первом случае нужно сначала выполнить команду `SET SQL_L0G_BIN=0` в процессе, который уничтожает данные на главном сервере, а уже потом приступать к удалению. Его достоинство заключается в том, что не требуется как-то специально конфигурировать репликацию на подчиненном сервере, а поскольку команды удаления даже не записываются в двоичный журнал, то и за счет этого эффективность хоть немного, да повышается. Основной же недостаток кроется в том, что двоичный журнал главного сервера невозможно использовать для аудита или восстановления на конкретный момент в прошлом, поскольку часть изменений, произведенных на сервере, в нем отсутствует. Кроме того, этот способ требует привилегии `SUPER`.

При использовании второго подхода нужно выполнить на главном сервере команду `USE`, то есть сделать текущей какую-то базу данных перед тем, как начинать удаление. Например, можно создать специальную базу с именем `purge`, указать в файле `my.cnf` на подчиненном сервере параметр `replicate_ignore_db=purge` и перезапустить его. В результате подчиненный сервер будет игнорировать все команды, выполненные в контексте текущей базы `purge`. У этого способа нет минусов, присущих предыдущему, но есть другой (не очень серьезный) недостаток: подчиненный сервер вынужден выбирать из двоичного журнала все события, даже те, в которых не заинтересован. Кроме того, есть опасность, что кто-то по ошибке выполнит в контексте базы данных `purge` запрос, не связанный с удалением архивных данных, и тогда этот запрос не будет воспроизведен на подчиненном сервере.

Утилита pt-archiver, входящая в комплект инструментов Percona Toolkit, поддерживает оба метода.

Есть и третий способ — воспользоваться параметром binlog_ignore_db для фильтрации событий репликации, но ранее мы отмечали, что считаем этот вариант слишком опасным.

Использование подчиненных серверов для полнотекстового поиска

Во многих приложениях необходимо сочетать транзакции с полнотекстовым поиском. Однако на момент написания книги возможность полнотекстового поиска существует лишь для таблиц типа MyISAM, а они не поддерживают транзак-

ции. (Предварительная версия полнотекстового поиска для InnoDB реализована в MySQL 5.6, но это еще не GA.) Типичное обходное решение заключается в том, чтобы сконфигурировать для полнотекстового поиска подчиненный сервер, поменяв на нем для некоторых таблиц подсистему хранения. Затем на подчиненном сервере можно построить полнотекстовые индексы. Тем самым удастся избежать потенциальных проблем, связанных с использованием в одном запросе транзакционных и нетранзакционных подсистем хранения, а заодно снять с главного сервера заботу о поддержании полнотекстовых индексов.

Подчиненные серверы только для чтения

Во многих организациях предпочитают использовать подчиненные серверы только для чтения, чтобы непреднамеренные изменения не нарушили репликацию. Этого можно добиться с помощью параметра `read_only`, который запрещает большинство операций записи, исключение составляют только потоки репликации, пользователи с привилегией `SUPER` и запись во временные таблицы. Эта тактика отлично работает при условии, что обычным пользователям не дают привилегию `SUPER` (этого не следует делать в любом случае).

Эмуляция репликации с несколькими главными серверами

В настоящее время MySQL не поддерживает репликацию с несколькими главными серверами, то есть наличие нескольких главных серверов для одного подчиненного. Однако ее можно эмулировать, поочередно изменяя главный сервер у подчиненного. Например, сначала вы назначаете в качестве главного сервер A, некоторое время работаете в такой конфигурации, потом делаете главным сервер B, снова какое-то время работаете, после чего опять возвращаетесь к серверу A. Насколько хороша такая схема, зависит от данных и от того, какой объем работы передают на подчиненный сервер оба главных. Если главные серверы не слишком загружены и выполняемые ими обновления не конфликтуют между собой, то все будет работать нормально.

Но придется позаботиться об отслеживании координат двоичного журнала обоих главных серверов. Кроме того, желательно, чтобы поток ввода/вывода на подчиненном сервере не извлекал больше данных, чем вы намереваетесь обрабатывать в каждом цикле, иначе можно значительно увеличить сетевой трафик, загружая и тут же выбрасывая большой объем информации на каждой его итерации.

Репликацию с несколькими главными серверами можно эмулировать также с помощью топологии «главный сервер — главный сервер» (или кольцо) с использованием подсистемы хранения Blackhole на подчиненном сервере (рис. 10.13).

В этой конфигурации оба главных сервера содержат свои собственные данные. Дополнительно на каждом созданы такие же таблицы, как на другом главном, но поскольку используется подсистема хранения Blackhole, то реальные данные в них не хранятся. Подчиненный сервер подключен к одному из главных — неважно, к какому именно. Но на подчиненном сервере подсистема хранения Blackhole не используется, поэтому фактически он подчинен обоим главным.

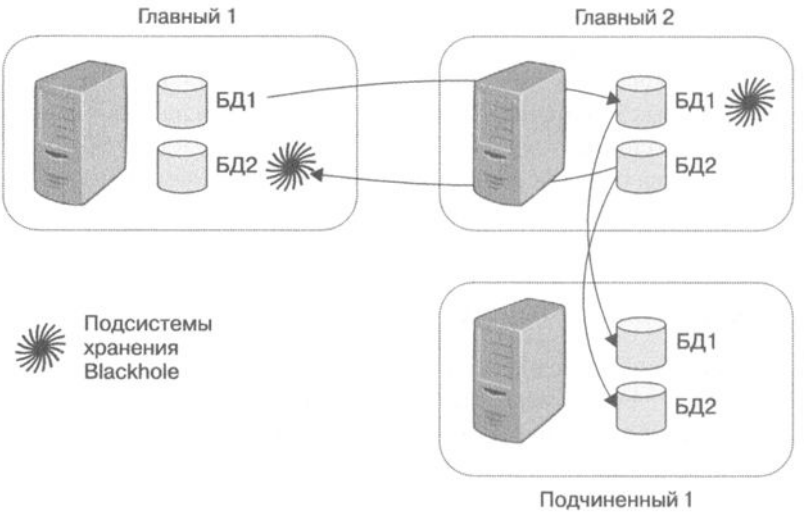


Рис. 10.13. Эмуляция репликации с несколькими главными серверами с применением двух главных серверов и подсистемы хранения Blackhole

На самом деле для достижения желаемого эффекта топология «главный сервер — главный сервер» не нужна. Можно просто реплицировать с `server1` на `server2` и далее на подчиненный. Если `server2` использует подсистему хранения Blackhole для таблиц, реплицированных с `server1`, то он не будет хранить данные, поступающие с `server1`. Такая схема изображена на рис. 10.14.

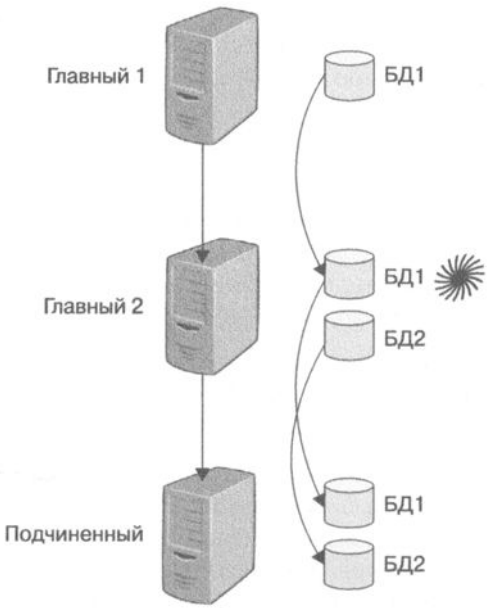


Рис. 10.14. Еще один способ эмулировать репликацию с несколькими главными серверами

Каждой из этих конфигураций свойственны общие проблемы, например конфликтующие обновления и команды `CREATE TABLE` с явно заданной подсистемой хранения.

Еще один вариант состоит в использовании Continuent Tungsten Replicator, о котором мы поговорим позже.

Создание сервера журналов

Среди прочего механизм репликации в MySQL позволяет создать сервер журналов, единственное назначение которого — упростить повтор и/или фильтрацию событий двоичного журнала. Далее в этой главе мы увидим, что это весьма полезно для возобновления репликации после сбоя. Кроме того, это полезно для восстановления данных на конкретный момент времени в прошлом, что мы обсудим в главе 15.

Представьте, что имеется набор двоичных журналов или журналов ретрансляции — быть может, из резервной копии, а может, с «упавшего» сервера — и вы хотите воспроизвести хранящиеся в них события. Можно было бы извлечь события с помощью утилиты mysqlbinlog, но удобнее и эффективнее просто настроить экземпляр MySQL вообще без данных и заставить его поверить, что это его собственные журналы. Чтобы создать временный сервер журналов, можно воспользоваться скриптом MySQL Sandbox (расположен по адресу http://mysqlsandbox.net). Серверу журналов не нужны никакие данные, потому что он будет не исполнять журналы, а только транслировать их другим серверам (однако учетная запись для репликации должна существовать).

Давайте посмотрим, как работает эта техника (примеры ее применения приведем далее). Предположим, что журналы называются `somelog-bin.000001`, `somelog-bin.000002` и т. д. Поместим эти файлы в каталог двоичных журналов на сервере журналов. Пусть это будет `/var/log/mysql`. Затем перед запуском сервера журналов изменим файл `my.cnf` следующим образом:

```
log_bin       = /var/log/mysql/somelog-bin
log_bin_index = /var/log/mysql/somelog-bin.index
```

Автоматически сервер не найдет файлы журналов, поэтому нужно также обновить индексный файл журналов сервера. В UNIX-системах это делается с помощью такой команды[1]:

```
# /bin/ls -1 /var/log/mysql/somelog-bin.[0-9]* > /var/log/mysql/somelog-bin.index
```

Убедитесь, что пользователь, от имени которого работает MySQL, может читать индексный файл и записывать в него. Теперь можно запустить сервер и с помощью команды `SHOW MASTER LOGS` проверить, что он видит файлы журналов.

Почему для восстановления сервер журналов предпочтительнее утилиты mysqlbinlog? По нескольким причинам.

- Репликация — это средство применения двоичных журналов, которое было протестировано миллионами пользователей и, как известно, работает. Никто

[1] Мы употребляем полное имя /bin/ls, чтобы не вызывалась программа с тем же именем, которая добавляет управляющие символы для выделения цветом на экране.

не может дать гарантию, что инструмент mysqlbinlog будет работать так же, как и репликация. Возможно, он не сможет точно воспроизвести изменения из бинарного журнала.

- Он быстрее, так как нет необходимости извлекать команды из журнала и по конвейеру передавать их процессу mysql.
- Легко наблюдать за ходом работы.
- Легко обходить ошибки. Например, можно пропускать команды, которые не удается реплицировать.
- Легко отфильтровывать события репликации.
- Иногда mysqlbinlog не сможет прочитать двоичный журнал из-за изменившегося формата.

Репликация и планирование производительности

Узким местом репликации являются операции записи, причем масштабированию они поддаются плохо. Планируя, какую долю общей пропускной способности системы отнимет добавление подчиненных серверов, нужно тщательно производить расчеты. Когда дело касается репликации, очень легко ошибиться.

Пусть, например, рабочая нагрузка состоит на 20 % из операций записи и на 80 % из операций чтения. Чтобы облегчить расчет, примем следующие очень приблизительные упрощающие предположения.

- Трудоемкость запросов на чтение и запись одинакова.
- Все серверы абсолютно идентичны между собой и способны обработать ровно 1000 запросов в секунду.
- Характеристики производительности главных и подчиненных серверов одинаковы.
- Все запросы на чтение можно переместить на подчиненные серверы.

Если в данный момент имеется один сервер, обрабатывающий 1000 запросов в секунду, то сколько подчиненных серверов придется добавить, чтобы можно было удвоить нагрузку и переместить на подчиненные серверы все запросы на чтение?

Может показаться, что нужно добавить два подчиненных сервера и распределить между ними 1600 запросов на чтение. Однако не будем забывать, что к рабочей нагрузке добавились 400 запросов на запись в секунду и распределить их между главным и подчиненными серверами не получится. Каждый подчиненный сервер должен выполнить 400 операций записи в секунду. Это означает, что он на 40 % занят записью и может обслужить лишь 600 запросов на чтение. Следовательно, для удвоения трафика нужно добавить не два, а три подчиненных сервера.

А что, если трафик снова удвоится? Теперь каждую секунду производится 800 операций записи, так что главный сервер пока справляется. Но каждый из подчиненных занят записью на 80 %, поэтому для обработки 3200 операций чтения потребуется

уже 16 подчиненных серверов. А если трафик еще чуть-чуть возрастет, то главный сервер может не справиться.

Как видите, масштабируемость далека от линейной: при увеличении количества запросов в четыре раза количество серверов возрастает в 17 раз. Это показывает, как быстро уменьшается отдача при добавлении серверов, подчиненных одному главному. И это еще при нереалистичных предположениях, в которых не учитывается, например, то, что при однопоточной покомандной репликации подчиненные серверы обычно имеют меньшую пропускную способность, чем главный. Поэтому в реальных условиях производительность репликации, скорее всего, будет еще хуже.

Почему репликация не помогает масштабированию записи

Основная проблема неудачного соотношения количества серверов и производительности состоит в том, что операции записи нельзя распределить между несколькими машинами так же равномерно, как операции чтения. Другими словами, репликация пригодна для масштабирования чтения, но не записи.

Возникает вопрос: можно ли как-то применить репликацию для повышения пропускной способности записи? Ответ: нет, ни в малейшей степени. Единственный способ масштабирования записи — секционирование данных, и мы обсудим его в следующей главе.

Возможно, у некоторых читателей возникла мысль использовать топологию «главный сервер — главный сервер» (см. подраздел «"Главный сервер — главный сервер" в режиме "активный — активный"» ранее в данной главе) и записывать на оба главных сервера. При такой конфигурации можно слегка увеличить количество операций записи по сравнению с топологией «главный — подчиненный», так как накладные затраты на сериализацию делятся пополам между обоими серверами. Если на каждом сервере выполняется 50 % операций записи, то сериализовать придется только те 50 %, которые обусловлены репликацией другого сервера. Теоретически это лучше, чем выполнять все 100 % записей параллельно на одной машине (главном сервере) и 100 % записей последовательно на другой машине (подчиненном сервере).

Идея может показаться заманчивой. Однако такая конфигурация не в состоянии обработать столько же операций записи, сколько один сервер. Сервер, для которого сериализуются 50 % операций записи, работает медленнее, чем сервер, выполняющий все операции записи параллельно.

Поэтому такая тактика не обеспечивает масштабирования записи. Это лишь способ разделить неэффективные сериализованные операции на оба сервера, так что самое слабое звено оказывается не таким уж слабым. Мы добьемся лишь сравнительно небольшого улучшения по сравнению с конфигурацией «активный — пассивный», увеличив риски и почти ничего не получив взамен. Более того, как показано в следующем разделе, в общем случае не будет даже минимального выигрыша.

Когда подчиненные серверы начнут отставать

Обычный вопрос, который задают о подчиненных серверах, звучит так: как предсказать, когда они не смогут справляться с изменениями, исходящими от главного сервера? Трудно заметить разницу между подчиненными серверами, работающими на 5 и 95 % от своей производительности. Тем не менее можно оценить производительность репликации и получить небольшое предупреждение о предстоящем насыщении.

В первую очередь нужно следить за пиками задержки. Если у вас есть график отставания репликации, можно заметить на нем небольшие кочки в те моменты, когда подчиненный сервер в течение короткого периода времени при возрастании объема работы не будет с ней справляться. По мере приближения рабочей нагрузки к полной загрузке подчиненного сервера эти кочки будут становиться все выше и шире. Передняя сторона кочки, как правило, будет иметь постоянный наклон, а наклон задней стороны, отражающий сокращение отставания подчиненного сервера, станет все меньше и меньше. Наличие этих кочек и их рост — это сигнал, предупреждающий о приближении к максимальной производительности подчиненного сервера.

Чтобы предсказать, что произойдет в какой-то момент в будущем, намеренно задержите работу подчиненного сервера, а затем посмотрите, как быстро он сможет нагнать. Цель этого действия состоит в определении крутизны задней поверхности кочки. Если вы остановите работу подчиненного сервера на час, затем запустите снова и он ликвидирует отставание за один час, значит, он работает наполовину от своей максимальной производительности. То есть, если вы остановите его в полдень и запустите снова в 13:00, а отставание будет ликвидировано к 14:00, это будет означать, что он за 1 час зафиксировал все изменения, произошедшие с 12:00 до 14:00. То есть за этот час он работал на двойной скорости.

Отметим, что в Percona Server и MariaDB вы можете напрямую измерить уровень использования репликации. Включите переменную сервера userstat, а затем выполните следующее:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.USER_STATISTICS
    -> WHERE USER='#mysql_system#'\G
*************************** 1. row ***************************
                 USER: #mysql_system#
      TOTAL_CONNECTIONS: 1
 CONCURRENT_CONNECTIONS: 2
        CONNECTED_TIME: 46188
             BUSY_TIME: 719
          ROWS_FETCHED: 0
          ROWS_UPDATED: 1882292
       SELECT_COMMANDS: 0
       UPDATE_COMMANDS: 580431
        OTHER_COMMANDS: 338857
   COMMIT_TRANSACTIONS: 1016571
 ROLLBACK_TRANSACTIONS: 0
```

Для того чтобы узнать, сколько времени поток репликации активно обрабатывал команды, сравните значение `BUSY_TIME` с половиной величины `CONNECTED_TIME`, по-

скольку на подчиненном сервере есть два потока репликации[1]. В нашем примере подчиненный сервер использует около 3 % своей производительности. Это не означает, что у него не будет случайных всплесков отставания, — если главный сервер выполнит изменение, которое займет 10 минут, вполне вероятно, что подчиненный отстанет примерно на такое же время при фиксации этого изменения. Однако это хороший индикатор того, что подчиненный сервер сможет восстановиться после любых испытываемых им всплесков нагрузки.

Запланированная недогрузка

Намеренная недогрузка серверов может оказаться наиболее правильным и экономичным способом построения крупномасштабных приложений, особенно если применяется репликация. Сервер, располагающий резервной производительностью, лучше выдерживает всплески нагрузки, обладает ресурсами для выполнения медленных запросов и операций обслуживания (например, команд `OPTIMIZE TABLE`) и лучше справляется с репликацией.

Попытка немного уменьшить затраты репликации за счет записи на оба узла в топологии «главный — главный», как правило, оказывается ложной экономией. Обычно любой сервер в паре нельзя загружать чтением более чем на 50 %, так как в противном случае в случае сбоя одного из них производительность окажется недостаточной. Если оба сервера способны справиться с нагрузкой самостоятельно, то, наверное, вообще нет смысла беспокоиться о затратах на однопоточную репликацию.

Кроме того, резервирование избыточной производительности — один из лучших способов обеспечить высокую доступность, хотя существуют и другие, например выполнение приложения в неполноценном режиме в случае отказа. В главе 12 эта тема будет рассмотрена подробнее.

Администрирование и обслуживание репликации

Вряд ли вам придется заниматься настройкой репликации постоянно, если только количество серверов не слишком велико. Но коль скоро репликация включена, то ее мониторинг и администрирование становятся регулярной задачей, сколько бы серверов ни было задействовано.

Эту работу следует максимально автоматизировать. Но писать собственные приложения для этой цели, возможно, и не понадобится: в главе 16 мы рассмотрим несколько инструментов повышения продуктивности для MySQL, средства мониторинга репликации уже встроены во многие из них или реализованы в виде подключаемых модулей.

---

[1] Если потоки репликации всегда работают, можете просто использовать время работы сервера, а не половину CONNECTED_TIME.

Мониторинг репликации

При использовании репликации сложность мониторинга MySQL возрастает. Хотя она производится как на главном, так и на подчиненном сервере, большую часть работы выполняет последний, и именно здесь чаще возникают проблемы. Все ли подчиненные серверы реплицируют данные? Были ли на каком-нибудь из них ошибки? Насколько отстал самый медленный из подчиненных серверов? MySQL предоставляет большую часть информации, необходимой, чтобы ответить на эти вопросы, но автоматизация процедуры мониторинга и обеспечение надежной репликации возлагаются на вас.

На главном сервере можно воспользоваться командой `SHOW MASTER STATUS`, которая покажет текущую позицию в двоичном журнале главного сервера и его конфигурацию (см. раздел «Конфигурирование главного и подчиненного серверов» ранее в данной главе). Можно также узнать, какие двоичные журналы есть на диске:

```
mysql> SHOW MASTER LOGS;
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| mysql-bin.000220 |    425605 |
| mysql-bin.000221 |   1134128 |
| mysql-bin.000222 |     13653 |
| mysql-bin.000223 |     13634 |
+------------------+-----------+
```

Эти сведения востребованы при определении параметров команды `PURGE MASTER LOGS`. Для просмотра событий репликации в двоичном журнале полезна команда `SHOW BINLOG EVENTS`. Например, после выполнения предыдущей директивы мы создали таблицу на неиспользуемом сервере. Поскольку точно известно, что это единственная команда, изменяющая данные, то мы знаем, что ее смещение в двоичном журнале равно `13634`, поэтому можно посмотреть, что туда было записано:

```
mysql> SHOW BINLOG EVENTS IN 'mysql-bin.000223' FROM 13634\G
*************************** 1. row ***************************
   Log_name: mysql-bin.000223
        Pos: 13634
 Event_type: Query
  Server_id: 1
End_log_pos: 13723
       Info: use `test`; CREATE TABLE test.t(a int)
```

Измерение отставания подчиненного сервера

Часто возникает необходимость следить за тем, насколько главный сервер опережает подчиненный. Хотя столбец `Seconds_behind_master` в таблице, которую выдает команда `SHOW SLAVE STATUS`, теоретически показывает отставание подчиненного сервера, фактически он не всегда точен по разным причинам.

- Подчиненный сервер вычисляет `Seconds_behind_master`, сравнивая текущую временную метку сервера с временной меткой, записанной в событии двоичного

журнала, поэтому подчиненный сервер не может даже сообщить о своем отставании, если не обрабатывает в данный момент какой-нибудь запрос.

- Обычно подчиненный сервер возвращает `NULL`, если на нем не работают процессы репликации.
- Некоторые ошибки (например, несоответствие конфигурационных параметров `max_allowed_packet` на главном и подчиненном серверах или неустойчивая сеть) могут нарушить репликацию и/или остановить потоки репликации на подчиненном сервере, но в столбце `Seconds_behind_master` при этом будет `0`, а не сообщение об ошибке.
- Иногда подчиненный сервер не в состоянии вычислить свое отставание, хотя процессы репликации работают. В этом случае он может вывести как `0`, так и `NULL`.
- Очень длительная транзакция может вызвать флуктуации отображаемого отставания. Например, если транзакция, обновляющая данные, остается открытой в течение часа, а затем выполняется коммит, то запись об обновлении попадет в двоичный журнал спустя час после того, как оно фактически состоялось. Когда подчиненный сервер приступит к обработке этой команды, он сообщит, что отставание от главного составляет час, а сразу вслед за этим — что никакого отставания нет.
- Если главный сервер-распространитель отстает, а ему подчинены какие-то серверы, то последние будут сообщать, что отставание составляет 0 секунд, если успевают за распространителем, хотя отставание от настоящего главного сервера ненулевое.

Все эти проблемы можно решить, если игнорировать величину `Seconds_behind_master` и судить об отставании подчиненного сервера с помощью чего-то, что можно наблюдать и измерять непосредственно. Неплохое решение дает *запись пульсации*, которая представляет собой временную метку, обновляемую на главном сервере один раз в секунду. Чтобы вычислить отставание, достаточно вычесть значение пульсации из текущей временной метки на подчиненном сервере. Этот метод лишен всех упомянутых недостатков и имеет дополнительное преимущество — удобную временную метку, показывающую, на какой момент времени актуальны данные на подчиненном сервере. Скрипт pt-heartbeat, входящий в состав комплекта инструментов Percona Toolkit, — наиболее популярная реализация репликации пульсации.

У пульсации есть и другие достоинства. Репликация записей пульсации в двоичном журнале полезна для многих целей, например аварийного восстановления, невозможного в других случаях.

Ни одна из рассмотренных метрик отставания не позволяет судить о том, сколько времени понадобится подчиненному серверу, чтобы догнать главный. Это зависит от целого ряда факторов, например от мощности подчиненного сервера и количества запросов, продолжающих поступать главному (см. подраздел «Когда подчиненные серверы начнут отставать» ранее в данной главе).

Как узнать, согласованы ли подчиненные серверы с главным

В идеальном мире подчиненный сервер всегда являлся бы точной копией главного. Но грубая реальность такова, что иногда из-за ошибок репликации данные на подчиненном и главном серверах могут рассинхронизироваться. Даже если явных ошибок нет, это все равно может произойти, поскольку некоторые операции MySQL реплицируются неправильно из-за ошибок в коде MySQL, искажений в сети, сбоев, некорректных остановов и т. д.[1]

Опыт показывает, что это правило, а не исключение, а значит, проверка согласованности подчиненных серверов с главными должна стать рутинной задачей. Особенно важно это, когда репликация применяется для резервного копирования, поскольку вряд ли вам захочется снимать резервную копию подчиненного сервера с искаженными данными.

В MySQL нет встроенного способа определить, совпадают ли данные на двух серверах. Зато имеются кое-какие заготовки для вычисления контрольной суммы таблиц и данных, например `CHECKSUM TABLE`. Однако сравнение подчиненного сервера с главным в момент, когда репликация работает, — задача нетривиальная.

В комплекте Percona Toolkit есть инструмент pt-table-checksum, который решает эту и ряд других задач. Его основная возможность состоит в проверке синхронизированности данных на главном и подчиненном серверах. Принцип работы основан на выполнении запросов `INSERT ... SELECT` на главном сервере.

В этих запросах вычисляется контрольная сумма данных, и результаты вставляются в таблицу. Затем указанные команды реплицируются и повторно выполняются на подчиненном сервере. Далее можно сравнить результаты, полученные на обоих серверах, и узнать, различаются ли данные. Поскольку вся процедура основана на репликации, то для получения правильных результатов не нужно одновременно блокировать таблицы на обоих серверах.

Типичный способ использования этого инструмента состоит в том, чтобы запустить его на главном сервере примерно с такими параметрами:

```
$ pt-table-checksum --replicate=test.checksum <master_host>
```

Данная команда вычисляет контрольные суммы всех таблиц и вставляет результаты в таблицу `test.checksum`. После того как эти запросы реплицированы на подчиненные серверы, можно выполнить простой запрос, который проверит, есть ли отличия от главного. Скрипт pk-table-checksum автоматически находит все серверы, подчиненные данному, запускает на каждом из них запрос и выводит результаты. На момент написания этой книги pt-table-checksum был единственным инструментом, который может достоверно сравнить данные реплики с данными на главном сервере.

---

[1] Если вы пользуетесь нетранзакционной подсистемой хранения, то останов сервера без предварительной команды STOP SLAVE считается некорректным.

Восстановление синхронизации подчиненного сервера с главным

Возможно, вы еще не один раз столкнетесь с рассинхронизацией подчиненного сервера. Может быть, расхождения будут выявлены в результате подсчета контрольных сумм, а может, вы знаете, что некий запрос был пропущен подчиненным сервером или что кто-то изменил данные непосредственно на нем.

Традиционно для восстановления синхронизации рекомендуют остановить подчиненный сервер и заново клонировать главный. Если несогласованность состояния критична, то, вероятно, стоит вывести подчиненный сервер из промышленной эксплуатации немедленно после обнаружения проблемы. Затем его можно заново клонировать или восстановить из резервной копии.

Недостаток этого подхода — неудобство, особенно когда объем хранимой информации велик. Если можно определить, какие данные различаются, то есть более эффективные способы, чем клонирование всего сервера. А если обнаруженная рассогласованность некритична, можно оперативно оставить подчиненный сервер и восстановить синхронизацию только затронутых данных.

Простейшая мера — выгрузить и снова загрузить рассогласованные объекты с помощью утилиты mysqldump. Этот подход вполне приемлем, если на протяжении данного процесса содержимое баз не изменяется. Тогда достаточно заблокировать таблицу на главном сервере, выгрузить ее, дождаться, пока подчиненный сервер догонит главный, и импортировать ее на подчиненный сервер (ждать нужно для того, чтобы не внести несогласованность в другие таблицы, например обновлявшиеся по результатам соединения с рассинхронизированной).

Во многих случаях такая тактика работает, однако к постоянно занятому серверу ее применить нельзя. Есть у нее и другой недостаток: данные на подчиненном сервере изменяются в обход репликации. Обычно безопаснее всего модифицировать информацию на подчиненном сервере посредством репликации, производя изменения на главном, поскольку при этом не возникает неприятных гонок и других сюрпризов. Кроме того, если объем данных очень велик или пропускная способность сети ограниченна, то выгрузка становится чересчур затратной. Что, если различаются только каждая тысячная строки в таблице, содержащей миллион строк? В этой ситуации выгрузка и загрузка всей таблицы — чрезмерное расточительство.

В комплекте Percona Toolkit есть инструмент pt-table-sync, решающий некоторые из описанных проблем. Он умеет эффективно находить и устранять различия между таблицами. К тому же его работа основана на механизме репликации, так что синхронизация подчиненного сервера достигается путем выполнения запросов на главном, что позволяет избежать гонок. Он интегрируется с таблицей контрольной суммы, созданной инструментом pt-table-check-sum, поэтому может работать с теми фрагментами таблиц, о которых известно, что они различаются. Впрочем, он пригоден далеко не всегда: для корректной работы необходимо, чтобы репликация

функционировала, поэтому в случае ошибки репликации рассчитывать на этот сценарий не приходится. Разработчики позаботились о максимальной эффективности утилиты pt-table-sync, но для очень больших объемов данных все равно нет смысла ее использовать. Сравнение терабайта данных на главном и подчиненном серверах неизбежно создает дополнительную нагрузку на оба. И все-таки в тех случаях, когда сценарий работает, он может сэкономить немало времени и сил.

Смена главного сервера

Рано или поздно наступает момент, когда подчиненному серверу требуется другой главный. Возможно, серверы ротируются для перехода на новую версию, или произошел сбой и бывший подчиненный сервер необходимо сделать главным, или просто требуется перераспределить производительность. Независимо от причины, подчиненному серверу нужно сообщить, что с этого момента назначенный ему главный сервер изменился.

Хорошо, когда процедура спланирована заранее, — так лучше, чем действовать в условиях стресса. Вам просто нужно выполнить команду `CHANGE MASTER` на подчиненном сервере, указав правильные параметры. Большинство параметров необязательны — можно задавать лишь те, которые нужно изменить. Подчиненный сервер отбросит текущую конфигурацию и журналы ретрансляции, после чего начнет реплицировать с вновь назначенного главного. Кроме того, новые параметры будут записаны в файл `master.info`, чтобы изменения не оказались потеряны после перезапуска.

Самая трудная часть процедуры — определить позицию в двоичном журнале нового главного сервера, чтобы подчиненный продолжил репликацию с той же логической точки, на которой остановился в момент внесения изменений.

Преобразование подчиненного сервера в главный несколько сложнее. Существует два основных сценария замены главного сервера одним из подчиненных ему. Первый — запланированное повышение, второй — незапланированное.

Запланированное повышение

Концептуально преобразовать подчиненный сервер в главный несложно. Далее перечислены шаги этой процедуры.

1. Остановить запись на старом главном сервере.
2. Дать возможность подчиненным серверам догнать главный (это не обязательно, но упрощает последующие шаги).
3. Сконфигурировать подчиненный сервер так, чтобы он стал новым главным.
4. Сообщить другим подчиненным серверам о новом главном и перенаправить на него операции записи.

Но проблема, как всегда, кроется в деталях. В зависимости от топологии репликации существуют различные ситуации. Так, описанные шаги выглядят несколько по-разному в конфигурациях «главный — главный» и «главный — подчиненный».

Опишем подробнее действия, которые, по всей вероятности, придется предпринять в большинстве конфигураций.

1. Прекратить любую запись на текущем главном сервере. Если возможно, принудительно остановить все клиентские программы, кроме соединений, связанных с репликацией. Неплохо, если в клиентские приложения встроен флаг Не работать, которым вы можете управлять. Если используются виртуальные IP-адреса, то можно попросту отключить виртуальный адрес, а затем прервать все клиентские соединения, чтобы закрыть открытые транзакции.
2. Запретить дальнейшие операции записи на главном сервере, выполнив команду `FLUSH TABLES WITH READ LOCK` (это не обязательно). Можно также перевести главный сервер в режим чтения, установив параметр `read_only`. Начиная с этого момента никакие операции записи на сервере, который вскоре будет заменен, невозможны. Ведь раз он больше не главный, любая запись на нем означала бы потерю данных! Однако обратите внимание на то, что установка параметра `read_only` не препятствует закоммичиванию транзакций. Для более надежной гарантии нужно прервать все открытые транзакции — это действительно остановит все записи.
3. Выбрать подчиненный сервер, которому суждено стать главным, и убедиться, что он полностью синхронизирован (позвольте ему закончить выполнение всех журналов ретрансляции, полученных от старого главного сервера).
4. Проверить, что новый главный сервер содержит в точности те же данные, что и старый (это необязательный шаг).
5. Выполнить на новом главном сервере команду `STOP SLAVE`.
6. Выполнить на новом главном сервере команду `CHANGE MASTER TO MASTER_HOST= ' '`, а затем `RESET SLAVE`, чтобы он отключился от старого главного и исключил информацию об этом соединении из своего файла `master.info`. (Это не будет работать правильно, если информация о соединении задана в файле `my.cnf`. В частности, поэтому мы рекомендуем не помещать ее туда.)
7. Получить координаты двоичного журнала нового главного сервера, выполнив команду `SHOW MASTER STATUS`.
8. Дождаться, пока все прочие подчиненные серверы догонят главный.
9. Остановить старый главный сервер.
10. В версии MySQL 5.1 и более поздних активизировать события на новом главном сервере, если это необходимо.
11. Позволить клиентам соединяться с новым главным сервером.
12. Выполнить на каждом подчиненном сервере команду `CHANGE MASTER`, указав на новый главный сервер. Использовать координаты двоичного журнала полученные при выполнении команды `SHOW MASTER STATUS`.

При повышении подчиненного сервера до уровня главного не забудьте удалить на нем все специфичные для подчиненных серверов базы данных, таблицы и привилегии. Кроме того, необходимо изменить все относящиеся к подчиненным серверам конфигурационные параметры, например нестрогий режим innodb_flush_log_at_trx_commit. Аналогично, если вы понижаете главный сектор до подчиненного, обязательно переконфигурируйте его по мере необходимости.

Если главные и подчиненные серверы сконфигурированы одинаково, то менять ничего не нужно.

Незапланированное повышение

Если произошел аварийный отказ главного сервера и его необходимо заменить подчиненным, процедура может осложниться. В случае, когда вы располагаете всего одним подчиненным сервером, выбирать не из чего. Но если таковых несколько, придется выполнить ряд дополнительных шагов.

Здесь добавляется проблема потенциальной потери событий репликации. Может случиться, что некоторые обновления, произведенные на главном сервере, еще не дошли ни до одного подчиненного. Возможно даже, что команда была выполнена на главном сервере, а затем отменена, но откат еще не реплицирован на подчиненный, — таким образом подчиненный сервер может опередить главный с точки зрения логической позиции репликации[1]. Если можно восстановить данные главного сервера на какой-то момент, то, может быть, вы сумеете найти потерянные команды и применить их вручную.

На всех последующих шагах в ходе вычислений следует использовать значения `Master_Log_File` и `Read_Master_Log_Pos`. Далее описана процедура повышения подчиненного сервера для топологии «главный — подчиненные».

1. Определить, какой из подчиненных серверов содержит наиболее актуальные данные. Выполнить на каждом из них команду `SHOW SLAVE STATUS` и выбрать тот, для которого координаты репликации `Master_Log_File/Read_Master_Log_Pos` самые свежие.
2. Дать всем подчиненным серверам закончить обработку журналов ретрансляции, полученных со старого главного сервера до момента его сбоя. Если сменить главный сервер до того, как подчиненный просмотрел весь журнал ретрансляции, оставшиеся в нем события будут отброшены и вы никогда не узнаете, на чем подчиненный сервер остановился.
3. Выполнить шаги 5–7 из списка, приведенного в предыдущем разделе.

[1] Это действительно возможно, хотя MySQL не журналирует события до коммита транзакции (подробнее см. в подразделе «Смешивание транзакционных и нетранзакционных таблиц» далее). Кроме того, это может произойти, когда главный сервер аварийно завершил работу и восстанавливается, но значение параметра innodb_flush_log_at_trx_commit отличалось от 1, поэтому сервер потерял некоторые изменения.

4. Сравнить координаты `Master_Log_File/Read_Master_Log_Pos` на каждом подчиненном сервере с соответствующими координатами на новом главном.
5. Выполнить шаги 10–12 из списка, приведенного в предыдущем разделе.

Мы предполагаем, что на всех подчиненных серверах включены параметры `log_bin` и `log_slave_updates`, как было рекомендовано в начале главы. В этом случае вы сможете восстановить все подчиненные серверы на один и тот же момент времени, что по-другому надежно сделать невозможно.

Нахождение нужных позиций в журнале

Если какой-нибудь подчиненный сервер находится не в той же позиции, что главный, то вам предстоит найти в двоичных журналах нового главного сервера позицию, соответствующую последнему событию, реплицированному на этот подчиненный сервер, и указать ее в команде `CHANGE MASTER TO`. Утилита mysqlbinlog позволяет узнать последний выполненный подчиненным сервером запрос и найти его в двоичном журнале нового главного сервера. Возможно, придется проделать кое-какие вычисления.

Чтобы проиллюстрировать сказанное, предположим, что номера событий журнала возрастают и что самый актуальный подчиненный сервер — новый главный — успел выбрать событие 100 как раз перед сбоем старого главного. Предположим также, что есть еще два подчиненных сервера, `replica2` и `replica3`, причем `replica2` извлек событие 99, а `replica3` — событие 98. Если для обоих подчиненных серверов указать текущую позицию в двоичном журнале нового главного, то они начнут репликацию с события 101, то есть произойдет рассинхронизация. Но если двоичный журнал нового главного сервера велся в режиме `log_slave_updates`, то вы сможете найти в нем события 99 и 100 и тем самым восстановить согласованное состояние подчиненных серверов.

Из-за перезапуска сервера, различий в конфигурации, ротации журналов или выполнения команды `FLUSH LOGS` смещение одного и того же события от начала журнала на разных серверах может различаться. Поиск событий иногда утомителен и занимает много времени, но обычно ничего сложного в нем нет. Достаточно просмотреть последнее событие, выполненное на каждом из серверов, запустив утилиту mysqlbinlog для двоичного журнала или журнала ретрансляции подчиненного сервера. Затем нужно отыскать тот же запрос в двоичном журнале нового главного сервера, опять же с помощью mysqlbinlog. В результате мы получим смещение запроса в байтах, которое можно подставить в команду `CHANGE MASTER TO`[1].

Эту процедуру можно ускорить, вычислив разность смещений событий, на которых остановились новый главный и подчиненный серверы. Если затем вычесть эту величину из текущей позиции в двоичном журнале нового главного сервера, то искомый запрос, скорее всего, будет находиться в получившейся позиции. Только не забудьте убедиться в этом, и если все так, то вы нашли позицию, с которой нужно запускать подчиненный сервер.

[1] Как уже упоминалось, записи пульсации из pt-heartbeat могут помочь в определении того, где именно в двоичном журнале вы должны искать свое событие.

Рассмотрим конкретный пример. Предположим, `server1` — главный сервер для `server2` и `server3`, причем `server1` вышел из строя. Исходя из значений `Master_Log_File/Read_Master_Log_Pos`, показанных командой `SHOW SLAVE STATUS`, делаем вывод, что `server2` реплицировал все события из двоичного журнала `server1`, но данные на `server3` еще неактуальны. Эта картина изображена на рис. 10.15 (события в журнале и байтовые смещения приведены только для иллюстрации).

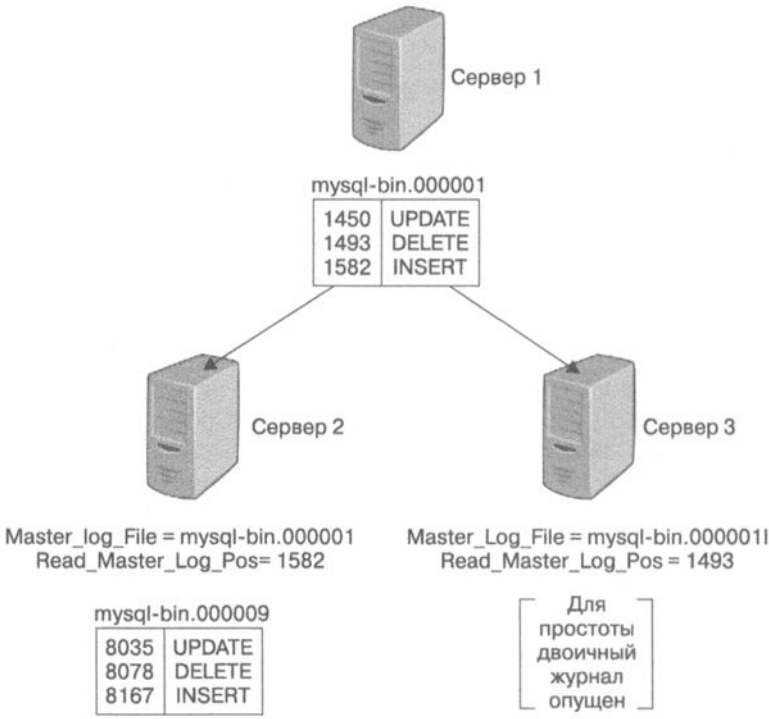


Рис. 10.15. В момент сбоя сервера 1 сервер 2 успел обработать все события, а сервер 3 отстал

Как видно из рис. 10.15, `server2` реплицировал все события из двоичного журнала главного сервера, так как значения `Master_Log_File` и `Read_Master_Log_Pos` на нем совпадают с последней позицией на `server1`. Поэтому `server2` можно сделать новым главным, а `server3` — его подчиненным.

Но какие параметры нужно указать в команде `CHANGE MASTER TO` на `server3`? Здесь придется немножко посчитать и подумать. Как видите, `server3` остановился на смещении 1493, которое на 89 байт отстает от смещения 1582, соответствующего последней команде, выполненной `server2`. В данный момент указатель записи в двоичном журнале на `server2` находится в позиции 8167; 8167 – 89 = 8078, поэтому теоретически мы должны указать `server3` именно эту позицию в двоичном журнале `server2`. Но все же следует посмотреть, какие события есть в окрестности этой позиции, и убедиться, что там действительно находится то, что нам нужно. Не исключено, что в этом месте оказалась другая команда, например, из-за обновлений данных, которые выполнялись только на `server2`.

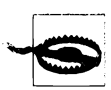
Предположим, что события совпадают, тогда сделать server3 подчиненным для server2 можно следующей командой:

```
server2> CHANGE MASTER TO MASTER_HOST="server2", MASTER_LOG_FILE="mysql-bin.000009",
         MASTER_LOG_POS=8078;
```

А что, если к моменту сбоя server1 уже закончил выполнение и журналирование еще одного события со смещением, большим 1582? Поскольку server2 успел прочитать и выполнить события только до смещения 1582, то одно событие можно потерять навсегда. Однако если диск старого главного сервера не поврежден, то отсутствующее событие все-таки удастся восстановить из его двоичного журнала с помощью утилиты mysqlbinlog или сервера журналов.

Если возникает необходимость восстановить отсутствующие события со старого главного сервера, то рекомендуем делать это после назначения нового главного, но до подключения к нему клиентов. В этом случае не придется вручную воспроизводить потерянные события на каждом подчиненном сервере — об этом позаботится механизм репликации. Но если вышедший из строя главный сервер вообще недоступен, то придется подождать и проделать эту работу позже.

Вариацией на ту же тему является надежное хранение двоичных журналов главного сервера, например в SAN-сети или на распределенном реплицируемом блочном устройстве (distributed replicated block device, DRBD). Даже если главный сервер полностью вышел из строя, его двоичные журналы никуда не денутся. Можно настроить сервер журналов, сделать его главным для всех подчиненных серверов и дать им возможность дойти то той точки, где главный сервер вышел из строя. После этого повышение любого из подчиненных серверов до главного довольно тривиально — по существу, оно ничем не отличается от запланированного повышения. В следующей главе мы рассмотрим различные системы внешней памяти.



При повышении подчиненного сервера до уровня главного не изменяйте его идентификатор. Сделав это, вы не сможете применить сервер журналов для воспроизведения событий старого главного сервера. Это одна из многих причин, по которым идентификаторы серверов следует рассматривать как неизменяемые атрибуты.

Смена ролей в конфигурации «главный сервер — главный сервер»

Одно из достоинств топологии репликации «главный — главный» состоит в том, что можно без труда поменять активный и пассивный серверы ролями, так как эта конфигурация симметрична. В этом разделе рассмотрим, как выполняется такое переключение.

При переключении ролей в данной конфигурации важнее всего гарантировать, что в любой момент запись производится только на один из двух серверов. Если она выполняется на оба сервера, то могут возникнуть конфликты. Иными словами, после

смены ролей пассивный сервер не должен получать события из двоичного журнала активного. Чтобы гарантировать такое положение вещей, вы должны подождать, пока поток репликации на подчиненном сервере подберет все события активного сервера, и только потом разрешать запись на нем.

Далее изложена последовательность действий, при которой смена ролей не грозит конфликтами в результате обновлений.

1. Прекратите все операции записи на активном сервере.
2. Выполните на активном сервере команду `SET @@global.read_only = 1` и установите параметр `read_only` в конфигурационном файле, чтобы изменение не было утрачено после перезапуска. Напомним, что это не помешает обновлять данные пользователям с привилегией `SUPER`. Если вы хотите запретить всем пользователям вносить изменения, выполните команду `FLUSH TABLES WITH READ LOCK`. В противном случае следует разорвать все соединения с клиентами, чтобы гарантированно не осталось долго работающих команд и незакоммиченных транзакций.
3. Выполните на активном сервере команду `SHOW MASTER STATUS` и запишите координаты репликации.
4. Выполните на пассивном сервере команду `SELECT MASTER_POS_WAIT()`, указав в ней координаты двоичного журнала активного сервера. Эта команда блокирует работу до тех пор, пока все подчиненные процессы не догонят активный сервер.
5. Выполните на пассивном сервере команду `SET @@global.read_only = 0`, тем самым превратив его в активный.
6. Измените конфигурацию приложений так, чтобы они вели запись на новый активный сервер.

В зависимости от особенностей приложений, возможно, придется выполнить и другие действия, например изменить IP-адреса обоих серверов. Поговорим об этом в следующих главах.

Проблемы репликации и их решение

Нарушить репликацию в MySQL совсем нетрудно. Простота реализации, позволяющая так легко настраивать механизм, означает одновременно, что его можно с той же легкостью остановить, запутать или иным способом вывести из строя. В этом разделе расскажем о типичных проблемах — о том, как они проявляются и как можно их разрешить или даже предотвратить.

Ошибки, вызванные повреждением или утратой данных

По разным причинам репликация в MySQL не очень устойчива к сбоям, исчезновению электропитания и повреждению данных, вызванному ошибками диска, памяти или сети. Почти наверняка из-за таких ошибок вам рано или поздно придется перезапускать репликацию.

Причиной многих проблем, появляющихся после неожиданного останова сервера, является то, что один серверов не сбросил что-то на диск. Далее описаны неприятности, с которыми можно столкнуться после неожиданного останова.

- *Неожиданный останов главного сервера.* Если главный сервер не сконфигурирован в режиме `sync_binlog`, то последние несколько событий перед сбоем могли быть не сброшены в двоичный журнал. Поэтому поток ввода/вывода на подчиненном сервере мог находиться в процессе чтения события, которое так и не попало на диск. После перезапуска главного сервера подчиненный повторно соединится с ним и попробует снова прочитать то событие, на котором остановился, однако получит ответ, что события с таким смещением нет. Передача данных из двоичного журнала главного сервера на подчиненный происходит практически мгновенно, поэтому подобную ситуацию нельзя назвать из ряда вон выходящей.

  Решение проблемы состоит в том, чтобы заставить подчиненный сервер начать чтение с начала следующего двоичного журнала. А чтобы события журнала не терялись, надо задать на главном сервере режим `sync_binlog`.

  Даже при установке режима `sync_binlog` данные в таблицах типа MyISAM могут быть повреждены при сбое, а транзакции подсистемы InnoDB — утрачены (но данные останутся неповрежденными), если параметр `innodb_flush_logs_at_trx_commit` не равен `1`.

- *Неожиданный останов подчиненного сервера.* Когда подчиненный сервер перезапускается после незапланированного останова, он читает свой файл `master.info`, чтобы понять, на чем прервалась репликация. К сожалению, этот файл не синхронизирован с диском, поэтому информация в нем может не соответствовать действительности. Подчиненный сервер, скорее всего, попытается повторно выполнить несколько событий, находящихся в двоичном журнале. Это чревато ошибками в связи с дублированием ключа в уникальном индексе. Если вы не можете определить, где на самом деле остановился подчиненный сервер (что маловероятно), то останется только проигнорировать такие ошибки. В этом может помочь инструмент pt-slave-restart, входящий в комплект Percona Toolkit.

- Если вы работаете только с таблицами InnoDB, то можете заглянуть в журнал ошибок MySQL после перезапуска подчиненного сервера. Процесс восстановления InnoDB выводит координаты двоичного журнала вплоть до точки восстановления. Их можно использовать, чтобы указать подчиненному серверу позиции в двоичном журнале главного. Percona Server дает возможность автоматически извлечь эту информацию во время процесса восстановления и обновления файла `master.info`, фактически передавая координаты репликации подчиненному серверу. MySQL 5.5 также предлагает параметры для управления тем, как `master.info` и другие файлы синхронизируются с диском, тем самым уменьшая эти проблемы.

Помимо потери данных из-за некорректного останова MySQL, не так уж редко случается повреждение двоичных журналов или журналов ретрансляции на диске. Перечислим наиболее типичные случаи.

- *Повреждение двоичных журналов на главном сервере.* Если повреждены двоичные журналы на главном сервере, то единственный выход — попробовать обойти испорченный фрагмент. Можно выполнить на главном сервере команду `FLUSH LOGS`,

чтобы он начал новый файл журнала, и указать подчиненному на его начало. Либо можно попытаться найти конец поврежденной области. Иногда полезна команда `SET GLOBAL SQL_SLAVE_SKIP_COUNTER = 1`, позволяющая пропустить одно поврежденное событие. Если таких событий несколько, попробуйте повторять эту команду, пока не пропустите их все. Но если повреждений много, вы, возможно, не сможете этого сделать: поврежденные заголовки событий могут помешать серверу найти следующее событие. В этом случае придется искать следующее неповрежденное событие вручную.

- *Повреждение журналов ретрансляции на подчиненном сервере.* Если двоичные журналы главного сервера не повреждены, то можно воспользоваться командой `CHANGE MASTER TO`, чтобы отбросить поврежденные журналы ретрансляции и попытаться получить их заново. Для этого нужно указать подчиненному серверу ту же позицию, на которой остановилась репликация (`Relay_Master_Log_File/Exec_Master_Log_Pos`). Это заставит его отбросить все имеющиеся на диске журналы ретрансляции. В MySQL 5.5 дела обстоят получше: она может автоматически обновлять журналы ретрансляции после сбоя.
- *Двоичный журнал рассинхронизирован с журналом транзакций InnoDB.* В случае сбоя главного сервера InnoDB может пометить транзакцию как закоммиченную, хотя она не попала в двоичный журнал на диске. Отсутствующую транзакцию восстановить невозможно, если только она не попала в журнал ретрансляции подчиненного сервера. Предотвратить такую ситуацию помогут параметр `sync_binlog` в MySQL 5.0 или пара параметров, `sync_binlog` и `safe_binlog`, в MySQL 4.1.

Если двоичный журнал поврежден, то количество данных, которое удастся восстановить из него, зависит от характера повреждения. Наиболее распространены следующие варианты.

- Байты изменены, но событие представляет собой допустимую SQL-команду. К сожалению, MySQL не способна даже обнаружить такое повреждение. Поэтому имеет смысл время от времени проверять корректность данных на подчиненных серверах. Видимо, это изменят в следующих версиях.
- Байты изменены, и событие представляет собой недопустимую SQL-команду. Возможно, удастся извлечь такое событие с помощью mysqlbinlog и увидеть искаженные данные, например:

  ```
  UPDATE tbl SET col????????????????
  ```

  Попробуйте найти начало следующего события, для чего нужно сложить смещение и длину, и напечатайте его. Быть может, удастся просто пропустить поврежденный фрагмент.
- Байты пропали и/или длина события неверна. В этом случае mysqlbinlog иногда завершается с ошибкой или даже аварийно, поскольку не может ни прочитать событие, ни найти начало следующего.
- Несколько событий повреждены или затерты либо произошло смещение данных в журнале, так что смещение следующего события указано неверно. И в этом случае mysqlbinlog мало чем поможет.

Если повреждение настолько серьезно, что mysqlbinlog не может прочитать события в журнале, то для поиска границ между событиями остается прибегнуть к шестнадцатеричному редактору или иному столь же трудоемкому методу. Но обычно особых трудностей не возникает, так как события разделены четко опознаваемыми маркерами.

Рассмотрим пример. Сначала взглянем, какие смещения событий в журнале показывает mysqlbinlog:

```
$ mysqlbinlog mysql-bin.000113 | egrep '^# at '
# at 4
# at 98
# at 185
# at 277
# at 369
# at 447
```

Простой путь поиска смещения в журнале заключается в их сравнении с результатом следующей команды `strings`:

```
$ strings -n 2 -t d mysql-bin.000113
       1 binpC'G
      25 5.0.38-Ubuntu_0ubuntu1.1-log
      99 C'G
     146 std
     156 test
     161 create table test(a int)
     186 C'G
     233 std
     243 test
     248 insert into test(a) values(1)
     278 C'G
     325 std
     335 test
     340 insert into test(a) values(2)
     370 C'G
     417 std
     427 test
     432 drop table test
     448 D'G
     474 mysql-bin.000114
```

Существует легко распознаваемый паттерн, позволяющий отыскивать начало события. Обратите внимание на то, что строки, оканчивающиеся буквой `G`, отстоят от начала события на один байт. Они являются частью заголовка события, имеющего фиксированную длину.

Точное значение зависит от исследуемого сервера. После небольшого расследования вы сможете обнаружить этот паттерн в своем двоичном журнале и определить смещение следующего неповрежденного события. После этого можно попытаться пропустить испорченные события, задав в командной строке mysqlbinlog аргумент `--start-position` или указав параметр `MASTER_LOG_POS` в команде `CHANGE MASTER TO`.

Использование нетранзакционных таблиц

Пока все идет нормально, покомандная репликация, как правило, отлично работает с нетранзакционными таблицами. Но если при обновлении нетранзакционной таблицы произошла ошибка, например команда была прервана до того, как завершилась, то на главном и подчиненном серверах могут оказаться разные данные.

Предположим, вы обновляете таблицу типа MyISAM, содержащую 100 строк. Что произойдет, если 50 строк обновились, а потом команда была прервана? Половина строк изменится, половина — нет. В результате реплика окажется рассинхронизирована, так как команда будет повторена на подчиненном сервере, где обновит все 100 строк (MySQL обнаружит, что команда завершилась с ошибкой на главном сервере, но не на подчиненном, и репликация остановится с сообщением об ошибке).

Если вы работаете с таблицами типа MyISAM, не забывайте выполнять команду `STOP SLAVE` перед остановом сервера, иначе в ходе останова будут прерваны все выполняющиеся запросы, в том числе незавершенные команды обновления. У транзакционных подсистем хранения такой проблемы не возникает. Незавершенное обновление транзакционной таблицы на главном сервере откатится, и информация о нем не попадет в двоичный журнал.

Смешивание транзакционных и нетранзакционных таблиц

При использовании транзакционной подсистемы хранения MySQL не записывает команды в двоичный журнал, пока транзакция не закоммичена. Следовательно, в случае отката транзакции соответствующие команды не журналируются и не повторяются на подчиненном сервере.

Но если команда затрагивает одновременно транзакционные и нетранзакционные таблицы и происходит откат, то MySQL откатывает лишь изменения в транзакционных таблицах, а изменения в нетранзакционных сохраняются. Пока не возникают ошибки, например прерванное обновление, никакой проблемы это не составляет: вместо того чтобы вообще не журналировать выполненные команды, MySQL заносит в журнал сначала их, а потом команду `ROLLBACK`. В результате на подчиненном сервере выполняются те же самые команды, что и на главном, и все хорошо. Конечно, это не слишком эффективно, так как подчиненный сервер должен проделать некую работу только для того, чтобы отбросить ее результат, но хотя бы теоретически синхронизация между подчиненным и главным серверами сохраняется.

Но все хорошо только до поры. Проблема появляется, когда на подчиненном сервере возникает взаимоблокировка, которой не было на главном. Изменения в таблицах транзакционного типа на подчиненном сервере откатываются, а в нетранзакционных остаются. В итоге данные на подчиненном и главном серверах различаются.

Единственный способ предотвратить такую ситуацию — не смешивать транзакционные и нетранзакционные таблицы в одной транзакции. Столкнувшись с подобной

проблемой, вы можете сделать только одну вещь — игнорировать ошибку на подчиненном сервере и заново синхронизировать рассогласованные таблицы.

При построчной репликации данной проблемы не возникает. В этом случае журналируются изменения строк, а не SQL-команды. Если команда изменяет какие-то строки в MyISAM-таблице и в InnoDB-таблице, а затем на главном сервере возникает взаимоблокировка и изменения в InnoDB-таблице откатываются, то изменения в MyISAM-таблице все равно записываются в двоичный журнал и воспроизводятся на подчиненном сервере.

Недетерминированные команды

Любая команда, изменяющая данные недетерминированным образом, может привести к расхождению информации, хранящейся на главном и подчиненном серверах. Например, команда `UPDATE` с ключевым словом `LIMIT` зависит от порядка выборки строк из таблицы. Если не гарантируется, что этот порядок на главном и подчиненном серверах совпадает, — скажем, если строки упорядочены по первичному ключу, — то команда может обновить на двух серверах разные строки. Такого рода ошибки выявить непросто, поэтому некоторые пользователи придерживаются правила не использовать `LIMIT` в командах, которые изменяют данные. Другим неожиданным источником недетерминированного поведения является команда `REPLACE` или `INSERT IGNORE` на таблице с более чем одним уникальным индексом — «победители» на главном и подчиненном серверах могут различаться.

Обращайте также внимание на команды, в которых участвуют таблицы из схемы `INFORMATION_SCHEMA`. Они вполне могут быть различными на главном и подчиненном серверах, что опять-таки приведет к расхождению. Наконец, имейте в виду, что большинство серверных переменных, например `@@server_id` и `@@hostname`, в версиях до MySQL 5.1 реплицируются некорректно.

У построчной репликации таких ограничений нет.

Использование различных подсистем хранения на главном и подчиненном серверах

Мы уже упоминали, что на подчиненном сервере часто бывает удобно использовать не ту подсистему хранения, что на главном. Однако в некоторых случаях это приводит к тому, что покомандная репликация порождает на подчиненном сервере иные данные. Например, вероятность того, что недетерминированные команды (в частности, упомянутые в предыдущем разделе) приведут к рассогласованию, растет.

Если вы обнаружите, что некоторые таблицы рассогласованы, то проверьте, какие подсистемы хранения используются для них на обоих серверах, а также изучите все запросы на обновление, в которых участвуют эти таблицы.

Изменение данных на подчиненном сервере

Покомандная репликация полагается на то, что данные на главном и подчиненном серверах одинаковы, поэтому не следует вносить или позволять изменения на подчиненном сервере (это легко гарантировать с помощью конфигурационной переменной `read_only`). Рассмотрим следующую команду:

```
mysql> INSERT INTO table1 SELECT * FROM table2;
```

Если таблица `table2` на подчиненном сервере содержит не те же данные, что на главном, то после выполнения этой команды и таблица `table1` станет отличаться. Другими словами, рассогласование одних таблиц имеет тенденцию перекидываться на другие. Данное утверждение относится ко всем типам запросов, а не только к `INSERT ... SELECT`. Окончиться это может двумя способами: либо вы получите на подчиненном сервере сообщение о нарушении ограничения уникальности, либо не получите никакого сообщения. Оповещение об ошибке — это удача, так как вы хотя бы будете знать, что произошла рассинхронизация подчиненного сервера с главным. Если же ошибка не сопровождается сообщением, хаос неминуем.

Единственный способ устранить последствия ошибки — заново синхронизировать данные с имеющимися на главном сервере.

Неуникальный идентификатор сервера

Это одна из самых трудноуловимых ошибок, с которыми можно столкнуться при репликации. Если случайно присвоить двум подчиненным серверам один и тот же идентификатор, то может показаться, что все работает нормально. Однако стоит внимательно приглядеться к журналам ошибок или понаблюдать за главным сервером с помощью утилиты innotop, как обнаружатся странные вещи.

Видно, что в каждый момент времени к главному серверу подключен только один из двух подчиненных (обычно подчиненные серверы подключены одновременно и все время занимаются репликацией). В журнале ошибок на подчиненном сервере вы увидите частые сообщения о разрыве и восстановлении соединения, но не найдете никакого упоминания о неправильно сконфигурированном идентификаторе.

В зависимости от версии MySQL подчиненные серверы могут реплицировать правильно, но медленно, или вовсе неправильно — любой из них может пропускать события в двоичном журнале или, наоборот, дважды обрабатывать одно и то же событие, что способно привести к нарушению ограничения уникальности (или к искажению данных без всяких сообщений). Возможны даже сбой или повреждение данных на главном сервере из-за возрастания нагрузки вследствие борьбы подчиненных серверов между собой. А если эта борьба становится особенно ожесточенной, то за очень короткое время журналы ошибок могут вырасти до необъятных размеров.

Единственное решение этой проблемы — быть очень внимательными при настройке подчиненных серверов. Возможно, стоит вести список идентификаторов, присвоен-

ных подчиненным серверам, чтобы не забыть, какой идентификатор какому из них назначен[1]. Если все подчиненные серверы находятся в одной подсети, то в качестве идентификатора можно выбрать последний октет IP-адреса.

Неопределенный идентификатор сервера

Если в файле `my.cnf` не определен идентификатор сервера, то MySQL позволит настроить репликацию командой `CHANGE MASTER TO`, но не даст запустить подчиненный сервер:

```
mysql> START SLAVE;
ERROR 1200 (HY000): The server is not configured as slave; fix in config file or with
CHANGE MASTER TO
```

Эта ошибка вызывает особенно сильное недоумение, если вы только что выполнили команду `CHANGE MASTER TO` и проверили настройки командой `SHOW SLAVE STATUS`. Команда `SELECT @@server_id` вернет какое-то значение, но это всего лишь значение по умолчанию. Вы должны задать идентификатор явно.

Зависимости от нереплицируемых данных

Если на главном сервере имеются базы данных или таблицы, отсутствующие на подчиненном, или наоборот, то по неосторожности можно легко нарушить репликацию. Предположим, что главный сервер хранит таблицу `scratch`, которой не существует на подчиненном. Если на главном сервере будет выполнена любая команда обновления с участием данной таблицы, то репликация нарушится, так как подчиненный сервер не сможет эту команду повторить. Аналогично, если вы создаете на главном сервере таблицу, которая уже существует на подчиненном, репликация будет прервана.

У проблемы нет обходного решения. Избежать ее можно только одним способом: не создавать на главном сервере таблиц, которых нет на подчиненном.

Как вообще появляются такие таблицы? Вариантов много, и некоторые предотвратить довольно трудно. Допустим, к примеру, что база данных `scratch` изначально была создана только на подчиненном сервере, а затем по какой-то причине главный и подчиненный серверы поменялись ролями. После этого вы, видимо, забыли удалить базу `scratch` и ее привилегии. Далее кто-то подключается к новому главному серверу и запускает в этой базе данных запрос или периодическое задание обнаруживает в ней таблицы и для каждой из них выполняет команду `OPTIMIZE TABLE`.

Об этом надо помнить, когда вы меняете подчиненный сервер на главный или думаете о том, как сконфигурировать подчиненные серверы. Любое отличие подчиненного сервера от главного — источник потенциальных проблем в будущем.

---

[1] Быть может, вам захочется сохранить его в таблице базы данных? Это лишь отчасти шутка — вполне можно добавить уникальный индекс по столбцу ID.

Отсутствующие временные таблицы

Временные таблицы очень удобны для ряда задач, но, к сожалению, несовместимы с покомандной репликацией. В случае сбоя или останова подчиненного сервера все временные таблицы, которые использовались в потоке репликации, внезапно исчезают. После перезапуска сервера последующие команды, которые ссылаются на отсутствующие временные таблицы, завершаются ошибкой.

Не существует безопасного способа использования временных таблиц на главном сервере в сочетании с покомандной репликацией. Многие настолько трепетно относятся к временным таблицам, что убедить их в этом почти невозможно, но, увы, это так[1]. Пусть даже временная таблица существует очень короткое время, она все равно может воспрепятствовать останову и запуску подчиненных серверов и восстановлению после сбоя. Это верно даже в случае, если такая таблица используется в пределах лишь одной транзакции. (Применение временных таблиц на подчиненном сервере, где они могут оказаться удобными, вызывает меньше проблем, но только если подчиненный сервер одновременно не является главным.)

Если репликация останавливается, потому что подчиненный сервер не может найти временную таблицу после перезапуска, у вас есть всего два выхода. Можно игнорировать ошибку или вручную создать таблицу с такими же именем и структурой, как у пропавшей временной. В любом случае данные на подчиненном и главном серверах, скорее всего, окажутся различными, если существовал хотя бы один запрос на запись с участием временной таблицы.

Избавиться от временных таблиц не так трудно, как кажется. У них есть два очень полезных свойства.

- Временная таблица видна только в контексте того соединения, где была создана, поэтому не конфликтует с одноименными таблицами, которые были созданы в других соединениях.
- Временная таблица исчезает при закрытии соединения, поэтому ее не нужно удалять явно.

Эти свойства можно эмулировать, зарезервировав специальную базу данных для псевдовременных таблиц, которые на самом деле являются постоянными. Нужно лишь выбирать для таблиц уникальные имена. К счастью, сделать это довольно просто — достаточно приписать к имени таблицы идентификатор соединения. Например, вместо команды `CREATE TEMPORARY TABLE top_users(...)` можно написать `CREATE TABLE temp.top_ users_1234(...)`, где `1234` — значение, возвращенное функцией

[1] Мы встречали людей, которые упорно пытались всеми возможными способами обойти эту проблему. Однако способа, позволяющего сделать временные таблицы безопасными для покомандной репликации, не существует. Точка. Неважно, что именно вы считаете; мы доказали, что это не сработает.

`CONNECTION_ID()`. После того как приложение закончило работать с псевдовременной таблицей, ее можно удалить явно или поручить это специальному процессу очистки. Наличие идентификатора соединения в имени позволяет легко определить, какие таблицы больше не используются, — можно получить список активных соединений с помощью команды `SHOW PROCESSLIST` и сравнить их с идентификаторами в именах таблиц[1].

У использования постоянных таблиц вместо временных есть и другие преимущества. Например, это упрощает отладку приложения, так как данные, которыми манипулирует приложение, видны из другого соединения. При использовании временных таблиц эта возможность была бы недоступна.

Однако постоянные таблицы, в отличие от временных, привносят некоторые накладные затраты: на их создание уходит больше времени, поскольку соответствующие `.frm`-файлы необходимо сбрасывать на диск. Для ускорения этой процедуры можно отключить режим `sync_frm`, но это опасно.

Если вы все-таки используете временные таблицы, то перед остановом подчиненного сервера не забывайте проверять, что переменная состояния `Slave_open_temp_tables` равна `0`. Если вы перезапустите подчиненный сервер при ненулевом значении, то в будущем, скорее всего, получите проблемы. Правильная последовательность действий такова: выполнить команду `STOP SLAVE`, проверить переменную и только потом останавливать сервер. Проверка переменной до остановки процессов репликации чревата гонками.

Репликация не всех обновлений

Если вы неправильно используете команду `SET SQL_LOG_BIN=0` или не понимаете правил фильтрации репликации, то подчиненный сервер может не выполнить некоторые обновления, произведенные на главном. Иногда именно такая цель и ставится (для архивирования), но чаще это происходит непреднамеренно и приводит к плачевным последствиям.

Предположим, что имеется правило `replicate_do_db`, которое разрешает реплицировать только базу данных `sakila` на один из подчиненных серверов. Если выполнить на главном сервере следующие команды:

```
mysql> USE test;
mysql> UPDATE sakila.actor ...
```

то данные на подчиненном и главном серверах рассинхронизируются. Другие команды могут даже вызвать ошибку репликации из-за нереплицируемых зависимостей.

---

[1] Утилита pt-find, также из пакета Percona Toolkit, позволяет легко удалять псевдовременные таблицы, если задать аргументы --connection-id и --server-id.

Конкуренция, вызванная блокировками при выполнении SELECT в InnoDB

Обычно в InnoDB команды `SELECT` не приводят к блокировке, но в некоторых случаях блокировку все же требуется установить. В частности, команда `INSERT ... SELECT` по умолчанию блокирует все строки, которые читает из исходной таблицы. MySQL вынуждена ставить блокировки для того, чтобы эта команда давала тот же самый результат при повторении на подчиненном сервере. По существу, блокировка сериализует команду на главном сервере, что совпадает со способом ее выполнения на подчиненном.

Такое проектное решение может привести к конкуренции за блокировки и к таймаутам в ожидании блокировки. Чтобы несколько сгладить остроту проблемы, не держите транзакцию на главном сервере открытой дольше, чем необходимо, — это уменьшит время удержания блокировок.

Кроме того, можно разбивать длинные команды на более короткие. Это эффективный способ уменьшить конкуренцию за блокировки, и даже если сделать это трудно, попытаться все равно стоит. (Добиться этого довольно просто с помощью инструмента pt-archiver пакета Percona Toolkit.)

Еще один обходной путь — заменить на главном сервере команды `INSERT ... SELECT` последовательностью команд `SELECT INTO OUTFILE` и `LOAD DATA INFILE`. Работает такой способ быстро и не требует блокирования. Мы согласны, что это трюк, но иногда он бывает полезен. Самое трудное здесь — выбрать уникальное имя выходного файла, которого не должно существовать на момент начала работы, и стереть его по ее завершении. Для подбора уникального имени можно воспользоваться техникой работы с функцией `CONNECTION_ID()` и периодически запускать задание, которое будет удалять файлы после того, как завершатся соединения, в которых они были созданы (с помощью crontab в UNIX-системах или планировщика задач в Windows).

Возможно, вы попытаетесь отключить блокировки вовсе вместо использования описанных обходных путей. Так можно сделать, но в большинстве случаев это безрассудство, поскольку в результате подчиненный сервер может рассинхронизироваться с главным без отправки каких бы то ни было сообщений. Кроме того, двоичный журнал станет бесполезным для восстановления сервера. Если вы все же решите, что риск оправдан, то установите следующий конфигурационный параметр:

```
# ЭТО НЕБЕЗОПАСНО!
innodb_locks_unsafe_for_binlog = 1
```

Теперь результат выполнения команды может зависеть от данных, при чтении которых не ставились блокировки. Если какая-то другая команда модифицирует эти данные и завершится раньше первой, то при повторе тех же команд на подчиненном сервере могут получиться иные результаты. Это справедливо как для репликации, так и для восстановления на конкретный момент времени в прошлом.

Чтобы понять, как блокировка предотвращает подобный хаос, представьте, что есть две таблицы, одна пустая, а другая с единственной строкой, в которой хранится значение 99. Указанные данные обновляются в двух транзакциях. Транзакция 1 вставляет содержимое второй таблицы в первую, а транзакция 2 обновляет вторую (исходную) таблицу (рис. 10.16).

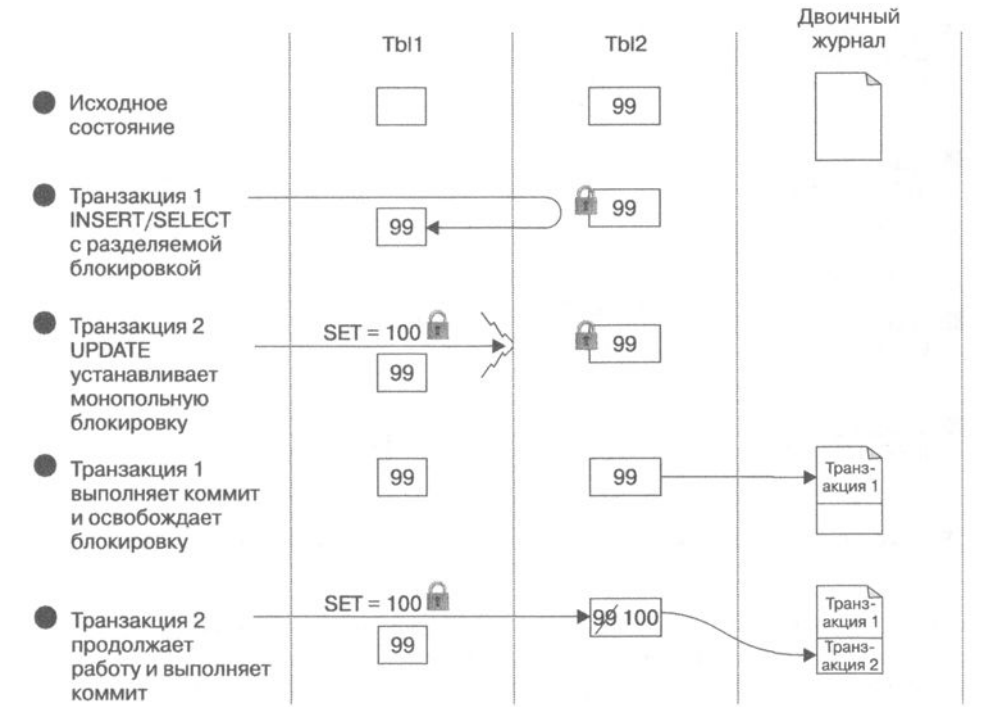


Рис. 10.16. Две транзакции обновляют данные, устанавливая разделяемые блокировки, чтобы сериализовать обновления

В этой последовательности действий очень важен шаг 2. На нем транзакция 2 пытается обновить исходную таблицу, для чего должна поставить монопольную блокировку (записи) на строки, которые собирается обновить. Монопольная блокировка несовместима ни с какой другой, в том числе с разделяемой блокировкой, поставленной на строку транзакцией 1, поэтому транзакция 2 должна дождаться коммита транзакции 1. Транзакции записываются в двоичный журнал в порядке коммита, поэтому их повтор на подчиненном сервере даст точно такие же результаты, как на главном.

В то же время если транзакция 1 не ставит разделяемую блокировку на строки, читаемые для команды `INSERT`, то никаких гарантий не дается. Изучите рис. 10.17, где показана возможная последовательность событий в случае, когда блокировки не ставятся.

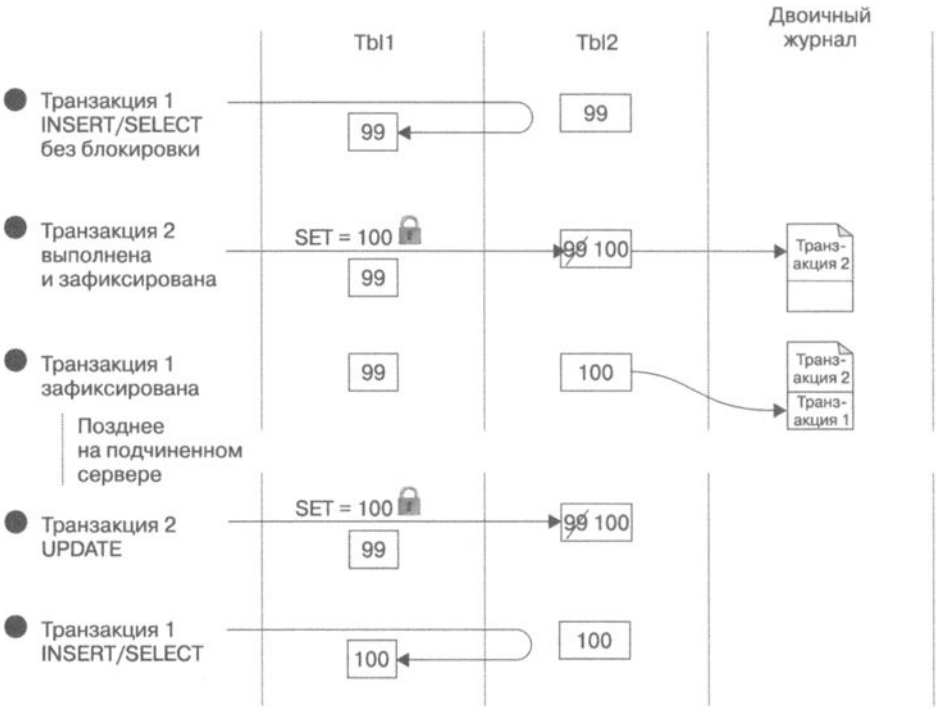


Рис. 10.17. Две транзакции обновляют данные, не устанавливая разделяемые блокировки для сериализации обновлений

Отсутствие блокировок позволяет записать транзакции в двоичный журнал в таком порядке, при котором повторное воспроизведение даст иные результаты, что и видно на рисунке. Сначала MySQL журналирует транзакцию 2, поэтому она повлияет на результаты выполнения транзакции 1 на подчиненном сервере. На главном ничего подобного не было. В итоге данные на главном и подчиненном серверах различаются.

Мы настоятельно рекомендуем в большинстве случаев оставлять параметр `innodb_locks_unsafe_for_binlog` равным `0`. Этой ситуации можно избежать при построчной репликации, при которой журналируются фактические изменения данных, а не команды.

Запись на обоих главных серверах в конфигурации «главный — главный»

Писать на оба главных сервера — ужасная идея. Часть возникающих при этом проблем можно разрешить безопасным способом, но не все. Но чтобы понять разницу между ними, требуется эксперт, набивший множество шишек.

В версии MySQL 5.0 появились два конфигурационных параметра, которые помогают разрешить конфликт между автоинкрементными первичными столбцами. Они называются `auto_increment_increment` и `auto_increment_offset`. Их можно

использовать для того, чтобы значения, генерируемые на разных серверах, чередовались, а не повторялись.

Однако это не решает всех проблем с выполнением записи на два главных сервера — так решается только проблема автоинкремента, а это лишь малая толика трудностей, с которыми придется столкнуться, если вы захотите записывать на оба сервера. Более того, это решение на самом деле порождает ряд новых проблем.

- Усложняется смена ролей серверов, участвующих в репликации.
- Из-за промежутков в нумерации напрасно расходуется пространство ключей.
- Оно работает только в том случае, когда для всех таблиц определены автоинкрементные первичные ключи (`AUTO_INCREMENT`), а делать это не всегда имеет смысл.

Вы можете самостоятельно генерировать неконфликтующие первичные ключи. Один из способов — создать первичный ключ по нескольким столбцам и в первом столбце хранить идентификатор сервера. Такой метод работает, но увеличивается длина первичного ключа, а в InnoDB это отражается еще и на внешних ключах.

В качестве альтернативы можно строить первичный ключ по одному столбцу, а идентификатор сервера хранить в старших битах целочисленного значения ключа. Реализуется это простыми операциями поразрядного сдвига влево (или умножения) и сложения. Например, если 8 старших бит беззнакового столбца типа `BIGINT` (64-разрядного) отведено под идентификатор сервера, то вставить значение `11` на сервере 15 можно так:

```
mysql> INSERT INTO test(pk_col, ...) VALUES( (15 << 56) + 11, ...);
```

Понять, что при этом происходит, проще всего, если перейти к основанию 2 и дополнить результат нулями до длины 64:

```
mysql> SELECT LPAD(CONV(pk_col, 10, 2), 64, '0') FROM test;
+------------------------------------------------------------------+
| LPAD(CONV(pk_col, 10, 2), 64, '0')                               |
+------------------------------------------------------------------+
| 0000111100000000000000000000000000000000000000000000000000001011 |
+------------------------------------------------------------------+
```

Но при таком подходе вам придется генерировать значения ключей самостоятельно, поскольку режим `AUTO_INCREMENT` отключен. Не используйте `@@server_id` вместо константы `15` в команде `INSERT`, поскольку тогда на подчиненном сервере получатся другие результаты.

Можно также прибегнуть к псевдослучайным значениям, используя такие функции, как `MD5()` или `UUID()`, но это негативно скажется на производительности, так как ключи получаются длинными и случайными, что плохо, в частности, для InnoDB. (Не пользуйтесь функцией `UUID()`, если только не генерируете значения в коде приложения, поскольку эта функция некорректно реплицируется в случае покомандной репликации.)

В общем, эта задача с трудом поддается решению, поэтому обычно мы рекомендуем перепроектировать приложение так, чтобы запись производилась только на один главный сервер. Кто бы мог подумать?

Слишком большое отставание репликации

Отставание репликации — распространенная проблема. Вне зависимости от причины рекомендуется проектировать приложение таким образом, чтобы оно сохраняло работоспособность при небольшом отставании подчиненных серверов. Если система не может работать в подобных условиях, то, по-видимому, ее архитектура не должна быть основана на репликации. Однако можно предпринять кое-какие шаги, чтобы подчиненные серверы не отставали от главного слишком намного.

Однопоточная природа репликации в MySQL означает, что на подчиненном сервере она принципиально менее эффективна. Даже очень быстрый подчиненный сервер с большим количеством дисков, процессоров и кучей памяти легко может отстать от главного, поскольку единственный поток обычно задействует лишь один процессор и диск. Но в любом случае подчиненный сервер должен быть не менее мощным, чем основной.

Блокировка на подчиненных серверах также представляет проблему. Прочие запросы, работающие на подчиненном сервере, могут заблокировать поток репликации. Поскольку репликация однопоточная, во время ожидания этот поток не может выполнять никакой полезной работы.

Отставание репликации может проявляться двояко: в виде кратковременных всплесков, после которых подчиненный сервер догоняет главный, и постоянного отставания. В первом случае запаздывание обычно вызвано одиночным долго работающим запросом, тогда как во втором никаких особенно длительных запросов может и не быть, а отставание все равно накапливается.

К сожалению, как уже говорилось, в настоящее время нелегко понять, насколько подчиненный сервер близок к исчерпанию своей производительности. Если нагрузка строго равномерна во времени, то подчиненный сервер будет одинаково хорошо работать вне зависимости от того, какая доля пропускной способности задействована — 10 или 99 %, но при достижении 100 % он внезапно начнет отставать. На практике нагрузка редко бывает равномерной, поэтому по мере приближения к пределу пропускной способности в пиковые моменты вы, вероятно, будете наблюдать, как отставание растет.

Журналирование запросов на подчиненном сервере и использование какого-нибудь инструмента анализа журналов, который может показать самые медленные запросы, — едва ли не лучшее, что можно сделать, когда подчиненный сервер перестает справляться. Не доверяйте интуитивным представлениям о том, что может работать медленно, и не основывайтесь на сведениях, как те или иные запросы выполняются на главном сервере, поскольку характеристики производительности на главном и подчиненном серверах сильно различаются. Самый лучший способ проделать такой анализ — временно включить журнал медленных запросов на подчиненном сервере и проанализировать его с помощью утилиты pt-query-digest, как было показано в главе 3. Стандартный журнал медленных запросов MySQL может журналировать запросы потока репликации в MySQL версии 5.1 и более поздних, если вы включите

параметр `log_slow_slave_statements`. Тем самым вы увидите, какие запросы во время репликации работают медленно. Percona Server и MariaDB позволяют включать и отключать этот параметр без перезапуска сервера.

Если подчиненный сервер не справляется с репликацией, то его настройка мало чего позволит добиться, положительного эффекта можно достичь, разве что установив более быстрые диски и процессоры (очень сильно могут помочь твердотельные диски, о них говорится в главе 9). Большинство рекомендаций связано с отключением чего-нибудь в попытке слегка уменьшить нагрузку на подчиненный сервер. Одно из простых решений — уменьшить частоту сброса на диск в подсистеме InnoDB, вследствие чего транзакции станут коммититься быстрее. Для этого присвойте параметру `innodb_flush_log_at_trx_commit` значение `2`. Можно также отключить на подчиненном сервере запись в двоичный журнал. Для этого в случае InnoDB нужно задать параметр `innodb_locks_unsafe_for_binlog` равным `1`, а в случае MyISAM — присвоить параметру `delay_key_write` значение `ALL`. Но за быстродействие приходится расплачиваться надежностью. Если будете повышать подчиненный сервер до уровня главного, не забудьте восстановить безопасные значения.

Не дублируйте затратные операции записи

Часто самый лучший способ помочь подчиненным серверам справиться с нагрузкой заключается в том, чтобы перепроектировать приложение и/или оптимизировать запросы. Попробуйте свести к минимуму объем работы, дублируемой в разных частях системы. Любая затратная операция записи на главном сервере должна быть повторена на каждом подчиненном. Если можно переместить часть работы с главного сервера на подчиненный, то выполнять ее придется только одному из подчиненных серверов. Затем результаты работы можно загрузить на главный сервер, например командой `LOAD DATA INFILE`.

Приведем пример. Предположим, что имеется очень большая таблица, которую необходимо агрегировать и свести к меньшей по размерам:

```
mysql> REPLACE INTO main_db.summary_table (col1, col2, ...)
    -> SELECT col1, sum(col2, ...)
    -> FROM main_db.enormous_table GROUP BY col1;
```

Если выполнять эту операцию на главном сервере, то каждый подчиненный вынужден будет повторить гигантский запрос с фразой `GROUP BY`. Когда подобных запросов станет много, подчиненные серверы начнут отставать. Поможет перенос запроса на какой-нибудь подчиненный сервер. На нем можно завести специальную базу, чтобы избежать конфликтов при обновлении данных, реплицируемых с главного сервера, и выполнить такой запрос:

```
mysql> REPLACE INTO summary_db.summary_table (col1, col2, ...)
    -> SELECT col1, sum(col2, ...)
    -> FROM main_db.enormous_table GROUP BY col1;
```

Теперь можно выполнить команду `SELECT INTO OUTFILE`, а вслед за ней команду `LOAD DATA INFILE` на главном сервере, чтобы загрузить в его базу данных результаты.

Вуаля — дублирование работы свелось к выполнению простейшей директивы `LOAD DATA INFILE`. Если имеется N подчиненных серверов, то мы сэкономили $N - 1$ больших запросов с фразой `GROUP BY`.

Впрочем, у этой стратегии есть и недостаток — неактуальные данные. Иногда бывает трудно получить согласованные результаты, выполняя чтение таблиц на подчиненном сервере и запись — на главном (эту проблему мы обсудим в следующих главах). Если чтение на подчиненном сервере вызывает сложности, то задачу можно упростить, сэкономив подчиненным серверам массу работы. Если отделить части `REPLACE` и `SELECT` друг от друга, то можно произвести выборку результатов в приложении, а затем вставить эти результаты обратно в базу данных на главном сервере. Сначала выполним на главном сервере следующий запрос:

```
mysql> SELECT col1, sum(col2, ...) FROM main_db.enormous_table GROUP BY col1
```

Затем вставим полученные результаты в сводную таблицу, повторив для каждой строки результирующего набора такой запрос:

```
mysql> REPLACE INTO main_db.summary_table (col1, col2, ...)  VALUES (?, ?, ...);
```

Как и раньше, нам удалось избавить подчиненные серверы от выполнения трудоемкой группировки. Разделение `SELECT` и `REPLACE` означает, что часть команд `SELECT`, которая отвечает за агрегацию, не повторяется на каждом подчиненном сервере.

Общая стратегия — избавление подчиненных серверов от затратной части операции записи — помогает во многих ситуациях, когда результаты запроса сложно получить, но после получения легко обработать.

Выполняйте параллельную запись без репликации

Еще один прием, помогающий сократить отставание подчиненных серверов, — обойти репликацию вовсе. Любая операция записи, выполненная на главном сервере, должна быть сериализована на подчиненном, поэтому сериализованную запись можно рассматривать как дефицитный ресурс. Задайтесь вопросом: все ли операции записи необходимо пропускать через репликацию от главного к подчиненному? Как сделать так, чтобы ограниченное количество ресурса «сериализованная запись» расходовалось только на те процедуры, которые действительно должны проходить через репликацию?

Такая постановка вопроса помогает определить приоритеты среди операций записи. В частности, если удастся выявить операции записи, легко выполняемые вне репликации, их можно распараллелить и сэкономить драгоценный ресурс подчиненного сервера.

Замечательным примером является архивирование, уже обсуждавшееся в этой главе. В OLTP-системах данные часто архивируются по строке за раз. Если просто переместить ненужные строки из одной таблицы в другую, то причина для репликации таких операций записи на подчиненных серверах может просто исчезнуть. Вместо этого можно отключить запись в двоичный журнал для команд архивирования, а за-

тем запустить отдельные идентичные процессы архивирования на главном и всех подчиненных серверах.

Мысль копировать данные на другой сервер самостоятельно вместо того, чтобы полагаться на репликацию, может показаться безумной, но в некоторых программах имеет смысл применять этот прием. В особенности тогда, когда приложение является единственным источником обновлений в некотором наборе таблиц. Часто узким местом является небольшое подмножество всех таблиц, и если именно их вывести из общей последовательности репликации, то можно существенно выиграть в скорости.

Инициализируйте кэш для потока репликации

Если позволяет рабочая нагрузка, то можно получить выигрыш от распараллеливания ввода/вывода на подчиненные серверы за счет предварительной загрузки данных в память. Эта техника не очень хорошо известна, но мы знаем, что она успешно применяется в некоторых крупных приложениях. В большинстве случаев использовать ее не придется, поскольку, если нет правильно сконфигурированного оборудования, а у рабочей нагрузки неподходящие характеристики, это будет безрезультатно. Другие типы изменений, которые мы только что обсуждали, обычно намного лучше и могут использоваться шире, чем вы думаете. Однако нам знакомо некоторое количество больших приложений, которые извлекают выгоду из предварительной выборки данных с диска.

Реализовать эту идею можно двумя способами. Первый состоит в том, чтобы использовать программу, которая немного опережает поток SQL в чтении данных из журналов ретрансляции и выполняет запросы в виде команд `SELECT`. В результате сервер считывает информацию с диска в память, так что к тому моменту, как поток SQL дойдет до соответствующей команды в журнале ретрансляции, ему не придется ждать выборки данных с диска. По существу, мы распараллеливаем ввод/вывод, который поток SQL обычно должен производить последовательно. Пока одна команда изменяет данные, в память с диска загружаются значения для следующей команды.

Этот прием работает только при следующих условиях.

- Поток SQL выполняет большой объем ввода/вывода, но в целом подчиненный сервер не перегружен этими операциями. Если это не так, то предварительная выборка не даст никакого выигрыша, поскольку диски сервера не простаивают.
- Подчиненный сервер оснащен большим количеством дисков — по восемь и более дисков на каждом подчиненном сервере.
- Используется подсистема хранения InnoDB, а рабочее множество настолько велико, что не помещается в памяти.

В качестве примера рабочей нагрузки, которая выигрывает от предварительной выборки, можно назвать большое количество команд `UPDATE`, обновляющих по одной строке в разных местах базы данных. Такая нагрузка обычно допускает высокую

степень конкуренции на главном сервере. Команды DELETE тоже можно ускорить таким образом. Для команд INSERT это менее вероятно, особенно если строки вставляются последовательно, поскольку конец индекса и так уже горячее место после предыдущих вставок.

Если по столбцам таблицы построено много индексов, то не всегда возможно предварительно выбрать все данные, которые команда может изменить. Команда UPDATE в состоянии модифицировать каждый индекс, тогда как SELECT обычно читает только первичный ключ и в лучшем случае один вторичный индекс. Поскольку UPDATE должна будет прочитать и другие индексы перед тем, как их модифицировать, тактика предварительной выборки оказывается не так эффективна.

Описанная техника не панацея. Существует много причин, по которым она может не дать эффекта и даже усугубить ситуацию. Применять ее следует лишь после тщательного изучения оборудования и операционной системы. Мы знаем людей, которым этот подход позволил увеличить скорость репликации на 300–400 %, но, когда попробовали его сами, у нас ничего не вышло. Очень важно правильно выставить параметры, но не всегда нужное сочетание настроек существует.

Инструмент mk-slave-prefetch, входящий в состав комплекта Maatkit, — один из пригодных для реализации идей, описанных в этом разделе. У него множество возможностей для работы в наибольшем числе случаев. Однако он довольно сложный и подходит только для опытных пользователей. Другим инструментом является slavereadahead Андерса Карлссона (Anders Karlsson), доступный по адресу http://sourceforge.net/projects/slavereadahead/.

Еще одна методика, разрабатываемая на момент написания книги, является внутренней для InnoDB. Она реализует транзакции в специальном режиме, который заставляет InnoDB подделывать обновления, поэтому процесс может произвести эти поддельные обновления, а затем поток репликации может быстро выполнить реальные обновления. Нечто подобное мы разрабатываем в Percona Server специально для очень популярного, масштабируемого в Интернете приложения. Проверьте, возможно, в настоящий момент оно уже разработано.

Если вы рассматриваете эту методику, мы полагаем, что лучше всего получить квалифицированный совет от эксперта, который знаком с тем, когда она работает и какие еще варианты доступны. Лучше всего оставить это напоследок, когда все остальное не получится.

Чрезмерно большие пакеты от главного сервера

Еще одна трудно обнаруживаемая проблема репликации связана с тем, что параметр max_allowed_packet на главном и подчиненном серверах не совпадает. В этом случае главный сервер может поместить в журнал пакет, который подчиненный сочтет слишком большим, и при попытке прочитать событие из двоичного журнала возникнут разного рода сбои. Это может, в частности, вызвать бесконечный цикл ошибок и повторных попыток чтения или повредить журнал ретрансляции.

Ограниченная пропускная способность сети

Если репликация производится по сети с ограниченной пропускной способностью, то можно включить на подчиненном сервере режим `slave_compressed_protocol` (реализован в версии MySQL 4.0 и более поздних). Когда подчиненный сервер соединяется с главным, он запрашивает сжатие — точно такое же, какое доступно любому клиенту MySQL. Для сжатия используется библиотека zlib, и наши тесты показывают, что текстовые данные иногда удается сжать до трети от первоначального размера. Но при этом требуется дополнительное процессорное время как на главном сервере для упаковки, так и на подчиненном — для распаковки.

Если канал на стороне главного сервера медленный, а на другом его конце много подчиненных серверов, то имеет смысл установить между главным и подчиненными серверами сервер-распространитель. Тогда с главным сервером по медленному каналу будет соединяться только один сервер, что уменьшит и потребление пропускной способности, и нагрузку на процессор главного сервера.

Отсутствие места на диске

Репликация может заполнить диски двоичными журналами, журналами ретрансляции и временными файлами, особенно если на главном сервере выполняется много команд `LOAD DATA INFILE`, а на подчиненном включен режим `log_slave_updates`. Чем сильнее отстает подчиненный сервер, тем больше места необходимо для журналов ретрансляции, которые уже прочитаны с главного сервера, но еще не обработаны. Чтобы предотвратить такие ошибки, следите за свободным местом на дисках и используйте параметр `relay_log_space`.

Ограничения репликации

Репликация в MySQL может остановиться или привести к рассинхронизации с сообщением об ошибке или без него, просто из-за присущих ей ограничений. Перечень функций SQL и приемов программирования, которые невозможно надежно реплицировать, довольно обширен (многие из них уже упоминались в данной главе). Трудно гарантировать, что ничего из этого списка не просочится в код промышленной системы, особенно если приложение большое или им занимается большой коллектив разработчиков[1].

Немало неприятностей причиняют и ошибки в коде сервера. Не хотим показаться критиканами, но ошибки встречались в коде репликации большинства основных версий MySQL, особенно в первых выпусках. А появление новых возможностей, в частности хранимых процедур, приводило к новым ошибкам.

[1] Увы, в MySQL отсутствует параметр forbid_operations_unsafe_for_replication (запретить операции, небезопасные для репликации). Однако в последних версиях она очень активно предупреждает об опасных процессах и даже отказывается от некоторых.

Для большинства пользователей это не причина сторониться новых возможностей. Это лишь повод для более тщательного тестирования, особенно при переходе на новую версию приложения или MySQL. Важен и мониторинг — проблема не должна оставаться незамеченной.

Репликация в MySQL устроена сложно, а чем сложнее приложение, тем тщательнее должно быть тестирование. Но если понять, как работать с репликацией, то все будет хорошо.

Насколько быстро работает репликация

Говоря о репликации, часто спрашивают: «Насколько быстро она работает?» Коротко можно ответить, что в общем случае настолько быстро, насколько MySQL успевает копировать события с главного сервера и воспроизводить их с очень небольшими накладными затратами. Если сеть медленная, а события в двоичном журнале очень велики, то задержка между записью в двоичный журнал и выполнением на подчиненном сервере может оказаться заметной. Если запросы выполняются долго, а сеть быстрая, то можно ожидать, что время выполнения запроса на подчиненном сервере будет составлять наибольшую долю общего времени, необходимого для репликации события.

Чтобы дать более точный ответ, нужно измерить все шаги процедуры и понять, на какие из них затрачивается больше всего времени. Некоторым читателям достаточно того, что обычно промежуток времени между журналированием событий на главном сервере и их копированием в журнал ретрансляции очень мал. Для тех же, кто любит детали, мы провели небольшой эксперимент.

Мы усовершенствовали процедуру, описанную в первом издании этой книги, и воспользовались методикой Джузеппе Максиа (Giuseppe Maxia)[1], для того чтобы с высокой точностью измерить скорость репликации. А еще написали недетерминированную пользовательскую функцию (UDF), которая возвращает системное время с точностью до микросекунд (ее исходный код приведен в разделе «Функции, определяемые пользователем» главы 7):

```
mysql> SELECT NOW_USEC()
+---------------------------+
| NOW_USEC()                |
+---------------------------+
| 2007-10-23 10:41:10.743917 |
+---------------------------+
```

Это позволило измерить скорость репликации, для чего мы вставляли значение `NOW_USEC()` в таблицу на главном сервере, а затем сравнивали его со значением на подчиненном сервере.

[1] См.: http://datacharmer.blogspot.com/2006/04/measuring-replication-speed.html.

Мы измеряли задержку, когда оба экземпляра MySQL были установлены на одной машине, чтобы избежать неточностей из-за расхождений в показаниях системных часов. Один экземпляр сконфигурировали как подчиненный другому, а затем выполнили на главном экземпляре следующие запросы:

```
mysql> CREATE TABLE test.lag_test(
    ->   id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->   now_usec VARCHAR(26) NOT NULL
    -> );
mysql> INSERT INTO test.lag_test(now_usec) VALUES( NOW_USEC() );
```

Мы воспользовались столбцом типа VARCHAR, потому что встроенные в MySQL типы для работы с временем не могут хранить временные данные с точностью более чем секунда (хотя некоторые функции умеют производить подобные вычисления). Остается лишь определить разность между моментами времени на главном и подчиненном серверах. Для этого отлично подойдет таблица типа Federated[1]. На подчиненном сервере был выполнен запрос

```
mysql> CREATE TABLE test.master_val (
    ->   id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->   now_usec VARCHAR(26) NOT NULL
    -> ) ENGINE=FEDERATED
    ->   CONNECTION='mysql://user:pass@127.0.0.1/test/lag_test',;
```

Простое соединение таблиц и функция TIMESTAMPDIFF() позволяют узнать, сколько микросекунд прошло между выполнением запроса на главном и подчиненном серверах:

```
mysql> SELECT m.id, TIMESTAMPDIFF(FRAC_SECOND, m.now_usec, s.now_usec) AS usec_lag
    -> FROM test.lag_test as s
    ->    INNER JOIN test.master_val AS m USING(id);
+----+----------+
| id | usec_lag |
+----+----------+
|  1 |      476 |
+----+----------+
```

Мы написали скрипт на языке Perl, которой вставляет 1000 строк в таблицу на главном сервере с 10-миллисекундной задержкой между последовательными вставками, чтобы предотвратить борьбу между главным и подчиненным экземплярами за процессорное время. Затем построили временную таблицу, содержащую задержки для каждого события:

```
mysql> CREATE TABLE test.lag AS
     > SELECT TIMESTAMPDIFF(FRAC_SECOND, m.now_usec, s.now_usec) AS lag
    -> FROM test.master_val AS m
    ->    INNER JOIN test.lag_test as s USING(id);
```

[1] Кстати, это единственный случай, когда кое-кто из авторов использовал подсистему хранения Federated.

Далее мы сгруппировали результаты по времени, чтобы узнать, какая задержка встречалась чаще всего:

```
mysql> SELECT ROUND(lag / 1000000.0, 4) * 1000 AS msec_lag, COUNT(*)
    -> FROM lag
    -> GROUP BY msec_lag
    -> ORDER BY msec_lag;
+----------+----------+
| msec_lag | COUNT(*) |
+----------+----------+
|   0.1000 |      392 |
|   0.2000 |      468 |
|   0.3000 |       75 |
|   0.4000 |       32 |
|   0.5000 |       15 |
|   0.6000 |        9 |
|   0.7000 |        2 |
|   1.3000 |        2 |
|   1.4000 |        1 |
|   1.8000 |        1 |
|   4.6000 |        1 |
|   6.6000 |        1 |
|  24.3000 |        1 |
+----------+----------+
```

Как видите, для большинства мелких запросов между выполнением на главном и подчиненном серверах проходит менее 0,3 миллисекунды.

Что мы не можем измерить таким образом, так это время доставки события на подчиненный сервер после того, как оно было записано в двоичный журнал главного. А хорошо бы его знать, поскольку чем раньше подчиненный сервер получит событие, тем лучше. Если событие получено, то подчиненный сервер будет иметь копию в случае сбоя главного.

Хотя измерения не показывают, сколько времени занимает эта часть процедуры, теоретически она должна выполняться чрезвычайно быстро (ограничена только скоростью сети). Процесс дампа двоичного журнала не опрашивает главный сервер о появлении новых событий — это было бы медленно и неэффективно. Вместо этого главный сервер уведомляет подчиненный о новых событиях. Чтение события из двоичного журнала главного сервера — это блокирующий системный вызов, который начинает посылать данные практически сразу после записи в журнал. Таким образом, можно с большой долей уверенности сказать, что событие прибывает на подчиненный сервер за то время, которое требуется, чтобы разбудить поток репликации на подчиненном сервере и передать информацию по сети.

Расширенные возможности репликации MySQL

В MySQL версии 5.5 Oracle значительно улучшил выполнение репликации, кроме того, много усовершенствований будет включено в релиз MySQL 5.6. Некоторые из них делают репликацию более надежной, другие добавляют многопоточную (параллельную) репликацию для смягчения проблем, связанных с существующим узким

местом (речь идет об однопоточности), а третьи увеличивают гибкость и улучшают контроль. Мы не будем много размышлять о функциональности, которая не относится к выпуску стабильной версии, однако есть несколько моментов, которые хотелось бы упомянуть, говоря об улучшениях, сделанных в MySQL 5.5.

В первую очередь нужно сказать о полусинхронной репликации, основанной на работе, которую Google выполнил несколько лет назад. Это, вероятно, самое большое изменение в этой сфере, начиная с версии MySQL 5.1, в которой была представлена построчная репликация. Такая репликация поможет вам убедиться, что на подчиненных серверах действительно есть копия данных главного сервера, поэтому вероятность потери данных в случае полного краха главного сервера уменьшилась.

Полусинхронная репликация добавляет задержку в процесс закоммичивания: при выполнении коммита транзакции события двоичного журнала должны быть переданы по меньшей мере одному подключенному подчиненному серверу до того, как клиентское соединение получит уведомление о завершении запроса. Эта задержка добавляется после того, как главный сервер выполнит коммит транзакции на свои диски. Таким образом, на самом деле это просто задержка для клиентов, которая не дает им выдавать множество транзакций на главный сервер быстрее, чем он может отправлять их на подчиненные.

Существует несколько распространенных заблуждений относительно полусинхронной репликации. Вот чего она не делает.

- Не блокирует выполнение коммита на главном сервере, пока подчиненные серверы не подтвердят получение. Коммит выполняется на главном сервере, и только уведомление клиента о коммите задерживается.
- Не блокирует клиента, пока подчиненные серверы не применили транзакцию. Подтверждение приходит после получения транзакции, а не после ее применения.
- Не обладает отказоустойчивостью. Если реплики не подтверждают получение, время ожидания постепенно истекает и происходит возврат к «обычному» режиму асинхронной репликации.

Тем не менее это очень полезный инструмент, который помогает гарантировать, что реплики действительно обеспечивают большую избыточность и надежность.

С точки зрения производительности полусинхронная репликация немного увеличивает запаздывание выполнения коммита, каким его видит клиент. Существует небольшая задержка из-за времени передачи по сети, времени, необходимого для записи и синхронизации данных с диском подчиненного сервера (если так сконфигурировано), и сетевого времени для подтверждения. Казалось бы, их можно сложить, однако в тестах это время оказалось едва заметным, вероятно, из-за того, что запаздывание скрыто другими причинами задержки. Джузеппе Максиа (Giuseppe Maxia) обнаружил приблизительно 200-секундную задержку производительности на каждый коммит[1]. Затраты будут более заметны по мере уменьшения размера транзакции.

[1] См.: http://datacharmer.blogspot.com/2011/05/price-of-safe-data-benchmarking-semi.html.

Фактически полусинхронная репликация может обеспечить достаточную гибкость для повышения производительности в некоторых случаях, сделав более безопасной возможность ослабить параметр `sync_binlog` на главном сервере. Запись в удаленную память (подтверждение от подчиненного сервера) происходит быстрее, чем запись на локальный диск (синхронизация при коммите). Хенрик Инго (Henrik Ingo) провел несколько сравнительных тестов, показавших повышение производительности при использовании полусинхронной репликации вместо строгой сохраняемости/устойчивости данных на ведущем узле[1]. В любой системе невозможно добиться абсолютной надежности — только более и более высоких ее уровней, и похоже, что полусинхронная репликация является наименее затратным способом повысить надежность данной системы, чем некоторые из альтернативных вариантов.

Помимо полусинхронной репликации, MySQL 5.5 поддерживает также репликацию пульсации, которая помогает подчиненным серверам оставаться соединенными с главным и избегать незаметного отсоединения. Если сетевое соединение будет разорвано, подчиненный сервер заметит отсутствие пульсации. Существуют улучшения, направленные на обработку разных типов данных между главным и подчиненным серверами, когда используется построчная репликация, а также несколько вариантов конфигурации того, как файлы метаданных репликации фактически синхронизируются с диском и как журналы ретрансляции обрабатываются после сбоя, что уменьшает вероятность возникновения проблем после сбоя и восстановления реплик.

Тем не менее мы еще не видели широкого промышленного развертывания каких-либо улучшений версии MySQL 5.5, направленных на репликацию, так что этот вопрос следует изучить подробнее.

Помимо сказанного приведем краткий перечень улучшений в работе либо MySQL, либо сторонних продуктов, таких как Percona Server и MariaDB.

- Oracle сделал множество улучшений в сборке MySQL 5.6 и промежуточных релизах.
  - Достигнуто состояние транзакционной репликации — отсутствуют файлы метаданных, которые могут рассинхронизироваться при сбое. (Некоторое время они существовали в Percona Server и MariaDB в другой форме.)
  - Вычисляются контрольные суммы двоичного журнала событий, помогающие обнаружить поврежденные события в журнале ретрансляции.
  - Реализована репликация с задержкой как замена инструмента pt-slave-delay из пакета Percona Toolkit.
  - Построчные двоичные журналы событий могут содержать исходный SQL-код, выполняемый на главном сервере.
  - Применяется многопоточная репликация (параллельная репликация).
- MySQL 5.6, Percona Server, заплатки Facebook и MariaDB имеют три разных варианта исправления проблем с групповыми коммитами, появившихся в MySQL 5.0.

[1] См.: http://openlife.cc/blogs/2011/may/drbd-and-semi-sync-shootout-large-server.

Прочие технологии репликации

Встроенная репликация — это не единственный способ репликации данных с одного сервера на другой, хотя, вероятно, лучше всего приспособленный для достижения большинства целей. (В отличие от PostgreSQL MySQL не имеет большого количества надстраиваемых возможностей репликации, возможно, потому, что встроенная репликация была добавлена на раннем этапе жизни продукта.)

Мы на короткой ноге с парочкой надстраиваемых технологий для репликации MySQL, например Oracle GoldenGate, но действительно недостаточно хорошо знакомы с большинством из них, чтобы много писать о них. Однако о двух хотелось бы упомянуть. Первая — это синхронная репликация Percona XtraDB Cluster, которую мы обсудим в главе 12, поскольку она лучше подходит для главы о высокой доступности. Вторая — Continuent's Tungsten Replicator (http://code.google.com/p/tungsten-replicator/).

Tungsten — продукт репликации промежуточного программного обеспечения с открытым исходным кодом, написанный на Java. Он имеет сходство с Oracle GoldenGate и, похоже, в будущих релизах получит множество сложных функций. На момент написания этой книги уже реализованы такие возможности, как репликация данных между серверами, автоматическое шардирование данных, параллельное применение изменений в подчиненных серверах (многопоточная репликация), повышение подчиненного сервера до главного при сбое последнего, кросс-платформенная и многосерверная репликация (несколько серверов реплицируются в один). Это версия с открытым исходным кодом пакета для кластеризации базы данных Tungsten Enterprise, которая является коммерческим программным обеспечением Continuent.

Кроме того, Tungsten реализует кластеры, состоящие из нескольких главных серверов, где записи могут быть направлены на любой сервер в кластере. Обобщенная реализация этой архитектуры требует обнаружения конфликтов и/или их разрешения. Это очень сложно, да и не всегда то, что нужно. Взамен Tungsten обеспечивает немного ограниченную реализацию, в которой не все данные доступны для записи на всех узлах, а каждый узел помечен как система записи для определенных битов данных. Это означает, что, например, офис в Сиэтле может записывать свои данные, которые реплицируются в Хьюстоне и Балтиморе. В Хьюстоне и Балтиморе данные доступны локально для чтения с малой задержкой, но Tungsten не дает их записывать, поэтому конфликтующие обновления невозможны. Хьюстон и Балтимор могут, конечно, обновлять собственные данные, которые также реплицируются во всех других местах. Такой подход к системе записи решает проблему, которую часто пытаются решить встроенной репликацией MySQL, известной как кольцо. Однако мы уже говорили, что она ни безопасна, ни надежна.

Tungsten Replicator не просто подключается и управляет репликацией MySQL — он заменяет ее. Он фиксирует изменения данных на серверах, читая их двоичные журналы. То же самое делает и встроенная функция MySQL, однако на этом ее функционал, в отличие от функционала Tungsten Replicator, заканчивается. Он считывает

двоичные журналы и извлекает транзакции, а затем выполняет их на подчиненных серверах.

Набор возможностей этого процесса намного богаче, чем у репликации MySQL. В частности, Tungsten Replicator стал первым продуктом, предлагающим параллельную репликацию для MySQL. Мы еще не видели этого в реальных условиях, но, как утверждается, в зависимости от характеристик рабочей нагрузки ожидается до трехкратного увеличения скорости репликации. С учетом всего того, что мы знаем об архитектуре продукта и о нем самом, эта информация кажется нам вполне достоверной.

Приведем список того, что нам нравится в Tungsten Replicator.

- Он обеспечивает встроенную проверку согласованности данных. Можно не продолжать.
- Он предлагает возможности подключаемого модуля, чтобы вы могли писать собственные пользовательские функции. Исходный код репликации MySQL очень трудно понять и еще сложнее изменить. Даже очень талантливые программисты наделали немало ошибок на сервере, пытаясь изменить код репликации. Приятно иметь возможность изменять репликацию без изменения кода репликации MySQL.
- Существуют глобальные идентификаторы транзакций, которые позволяют вам определять состояние серверов по отношению друг к другу, не пытаясь сопоставить имена и смещения двоичных журналов.
- Это хорошее решение с высокой доступностью, которое позволяет быстро повышать подчиненный сервер до главного.
- Он поддерживает гетерогенную репликацию (например, между MySQL и PostgreSQL или MySQL и Oracle).
- Он поддерживает репликацию между версиями MySQL в случаях, когда репликация MySQL не является обратно совместимой. Это очень хорошо для определенных сценариев обновления, где в противном случае вы, скорее всего, не сможете создать работоспособный сценарий отката, если обновление пойдет не так как хочется. Либо вам придется обновлять серверы не в том порядке, который вы бы предпочли.
- Проектирование параллельной репликации подходит для шардированных или многоклиентских приложений.
- Приложения Java могут без изменения кода записывать на главный сервер и читать из подчиненных.
- Он стал намного проще и легче в настройке и администрировании, чем раньше, во многом благодаря добросовестной работе директора по качеству Джузеппе Максиа (Giuseppe Maxia).

Приведем и ряд недостатков.

- Надо полагать, что он все же сложнее, чем встроенная репликация MySQL, поскольку включает в себя больше элементов, которые надо настраивать и администрировать. В конце концов, это промежуточное программное обеспечение.

- Это еще один элемент вашего стека приложений, который надо изучить и понять.
- Он не такой легковесный, как встроенная репликация MySQL, и не имеет такой хорошей производительности. Однопоточная репликация медленнее, чем однопоточная репликация MySQL.
- Он не так широко тестировался и развертывался, как репликация MySQL, поэтому выше риск возникновения ошибок и проблем.

В целом, мы рады, что Tungsten Replicator доступен и активно развивается, стабильно получая новые возможности и функциональность. Приятно иметь альтернативу встроенной репликации, тем самым MySQL становится подходящей для использования в большем количестве случаев и достаточно гибкой для удовлетворения требований, которые, по-видимому, никогда не сможет удовлетворить встроенная репликация MySQL.

Итоги главы

Репликация MySQL — это швейцарский армейский нож встроенных возможностей MySQL, она значительно расширяет диапазон функциональности и полезности MySQL. На самом деле это, возможно, одна из ключевых причин, почему MySQL так быстро стала настолько популярной.

Хотя репликация имеет множество ограничений и оговорок, выясняется, что большинство из них относительно неважны либо основная масса пользователей их легко преодолевает. Многие из недостатков — это просто особое поведение расширенных возможностей, которые большинство людей не используют, но которые очень полезны для нуждающегося в них меньшинства.

Репликация предлагает важный и сложный функционал, но при этом сам сервер не позволяет конфигурировать, контролировать, администрировать и оптимизировать все имеющиеся «примочки». Здесь огромную помощь могут оказать инструменты сторонних производителей. Мы субъективны, но считаем, что наиболее заметными инструментами для улучшения вашей жизни с помощью репликации являются Percona Toolkit и Percona XtraBackup. Советуем проверить их тестовые комплекты, прежде чем использовать какие-либо другие инструменты. Если у них нет официальных автоматизированных наборов тестов, задумайтесь, прежде чем доверять им свои данные.

Когда дело доходит до репликации, ваш девиз должен быть K.I.S.S[1]. Не делайте ничего необычного, например не используйте кольца репликации, таблицы Blackhole или фильтры репликации, если не уверены, что вам это действительно необходимо.

---

[1] «Будь проще, Шварц!» (Keep It Simple, Schwartz!) Кое-кто из нас считает, что K.I.S.S. означает именно это.

K.I.S.S. может расшифровываться по-разному. Например: «Пусть будет коротко и ясно» (Keep It Short and Simple) или «Будь проще, дурачок» (Keep It Simple, Stupid). — *Примеч. пер.*

Используйте репликацию просто для зеркалирования полной копии ваших данных, включая все привилегии. Поддержание подчиненного сервера полностью идентичным главному поможет вам избежать многих проблем.

Говоря о поддержании подчиненного сервера идентичным главному, приведем краткий список важных операций, которые нужно выполнять при использовании репликации.

- Применяйте утилиту pt-table-checksum из пакета Percona Toolkit для проверки того, являются ли реплики истинными копиями главного сервера.
- Контролируйте репликацию, чтобы убедиться, что она работает и не отстает от главного сервера.
- Не забывайте об асинхронном характере репликации и разрабатывайте приложение так, чтобы либо избежать чтения устаревших данных из подчиненного сервера, либо это было неважно.
- Не делайте записи на несколько серверов. Настройте подчиненные серверы как доступные только для чтения и заблокируйте привилегии для предотвращения изменения данных.
- Включите настройки безопасности и готовности к работе, как описано в этой главе.

Как будет сказано в главе 12, сбой при репликации является одной из наиболее распространенных причин простоев MySQL. Чтобы избежать проблем с репликацией, прочитайте эту главу и попытайтесь применить ее рекомендации на практике. Кроме того, внимательно изучите посвященный репликации раздел руководства по MySQL и узнайте, как она работает и как ее администрировать. Если вам нравится читать, книга *MySQL High Availability* (Charles Bell et al., издательство O'Reilly) также содержит полезную информацию о внутренних функциях репликации. Но все равно нужно прочитать инструкцию!

11 Масштабирование MySQL

В этой главе мы покажем, как на основе MySQL можно построить решение, которое может сильно увеличиться в размерах, оставаясь при этом быстрым, эффективным и экономичным.

Какой совет можно дать по поводу масштабируемости приложений, которые отлично подходят для одного или пары серверов? Большинство людей никогда не будут поддерживать чрезвычайно большие системы, поэтому тактику, используемую в очень больших компаниях, нельзя применять для любых компаний. В этой главе попытаемся охватить целый диапазон стратегий. Мы создали или помогли создать множество приложений, начиная с тех, которые используют один-два сервера, и заканчивая такими, которые используют тысячи. Выбор подходящей стратегии для вашего приложения часто является ключом к экономии денег и времени, которые можно было бы инвестировать куда-то еще.

MySQL часто критикуют за то, что она тяжело масштабируется. Иногда это так и есть. Однако вы можете удачно масштабировать MySQL, если выберете правильную архитектуру и хорошо ее реализуете. Но далеко не все правильно понимают, что такое масштабируемость, поэтому мы начнем с разъяснения этого понятия.

Что такое масштабируемость

В разговоре такие термины, как «масштабируемость», «высокая доступность» и «производительность», часто употребляют как синонимы, но на самом деле они обозначают совершенно разные вещи. Как мы писали в главе 3, под производительностью мы понимаем время отклика. Масштабируемость рассмотрим подробнее чуть позже, но в двух словах — это способность системы получать эффект от каждого вложенного рубля по мере того, как вы добавляете ресурсы для увеличения выполненной работы. Плохо масштабируемая система достигает точки минимума отдачи и не может расти дальше.

Мощность[1] — это родственная категория. Мощность системы — это объем работы, которую она может выполнить за определенный промежуток времени[2]. Однако мощность должна отвечать требованиям системы. Максимальная пропускная способность системы не то же самое, что ее мощность. Большинство эталонных тестов измеряют максимальную пропускную способность системы, но в реальности системы так не нагружают. Если вы это сделаете, производительность ухудшится, а время отклика станет неприемлемо большим и непостоянным. Мы определяем фактическую мощность системы как пропускную способность, которой она может достичь, обеспечивая приемлемую производительность. Вот почему результаты эталонных тестов обычно не должны сводиться к одному показателю.

Мощность и масштабируемость не связаны с производительностью. Проведем аналогию с машинами на шоссе.

- Производительность — это скорость, с которой может ехать автомобиль.
- Мощность — это ограничение скорости и количество автомобилей на автомагистрали.
- Масштабируемость — предел, до которого можно увеличивать количество автомобилей и полос движения без снижения скорости транспортного потока.

В этой аналогии масштабируемость зависит от таких факторов, как удобство развязок, количество поломок машин и дорожно-транспортных происшествий, а также интенсивность перестроения из ряда в ряд, но в общем случае не зависит от мощности двигателей. Это не означает, что производительность не имеет значения, поскольку это не так. Мы просто хотим сказать, что системы могут быть масштабируемыми, даже если не являются высокопроизводительными.

С высоты птичьего полета масштабируемость — это возможность увеличить производительность при увеличении ресурсов.

Даже если ваша архитектура MySQL масштабируема, приложение может таковым не являться. Если в силу каких-либо причин мощность увеличить сложно, приложение в целом не масштабируется. Минуту назад мы определили мощность с точки зрения пропускной способности, но стоит посмотреть на нее с высоты птичьего полета. С этой точки зрения мощность просто означает способность обрабатывать нагрузку, а ее также следует рассматривать с разных точек зрения.

- *Объем данных*. Рост объема данных, накапливаемых приложением, — одна из самых типичных проблем масштабирования. Особенно актуально это для многих современных веб-приложений, которые никогда не удаляют данные. На-

---

[1] В оригинале употреблен термин capacity, который здесь переводится как «мощность», но его следует отличать от power. — *Примеч. пер.*

[2] В академической физике мощность определяется как «работа, выполняемая в единицу времени». Но в вычислительной технике семантика слова «мощность» столь многогранна, что мы стараемся избегать этого термина из-за его многозначности. Поэтому при описании систем корректнее говорить о мощности (capacity) как о максимальном КПД, достижимом в данной системе.

пример, сайты социальных сетей обычно не удаляют старые сообщения и комментарии.

- *Количество пользователей.* Даже если каждый пользователь порождает небольшой объем данных, при увеличении их числа он суммируется. Помимо этого, общий объем данных может расти несоизмеримо быстрее, чем количество пользователей. Кроме того, обычно чем больше пользователей, тем больше транзакций, и зависимость между количеством транзакций и количеством пользователей тоже может быть нелинейной. Наконец, с увеличением числа пользователей (и объема данных) сложность запросов чаще всего возрастает, особенно если они зависят от количества связей между пользователями (количество связей ограничено сверху величиной $(N(N-1))/2$, где N — число пользователей).
- *Активность пользователей.* Не все пользователи одинаково активны, и их активность распределена неравномерно. Если внезапно активность пользователей возрастет, например, из-за появления новых возможностей, то нагрузка на приложение может значительно увеличиться. Активность пользователей не сводится к количеству просмотров страниц — при неизменном числе просмотров нагрузка может стать больше, если популярной окажется часть сайта, требующая большей работы для генерации страниц. Кроме того, некоторые пользователи гораздо активнее прочих: у них больше друзей, сообщений или фотографий, чем у среднего пользователя.
- *Размер взаимосвязанных наборов данных.* Если между пользователями имеются связи, то, возможно, приложению приходится выполнять запросы или вычисления для целых групп связанных пользователей. Это сложнее, чем работать с индивидуальными пользователями и их данными. Сайты социальных сетей нередко сталкиваются с непростыми проблемами из-за особо популярных групп или пользователей, у которых много друзей[1].

Формальное определение. Стоит изучить математическое определение термина «масштабируемость», поскольку это позволит вам понимать более сложные концепции. Без этого могут возникнуть проблемы с пониманием или точным определением этого термина. Не беспокойтесь, это не связано с высшей математикой — вы сможете понять эту концепцию интуитивно, даже не будучи математическим гением.

Ключевым аспектом является фраза, которую мы использовали ранее: «Получать эффект на каждый вложенный рубль». Другими словами масштабируемость можно определить как уровень, при котором система обеспечивает одинаковую рентабельность инвестиций (return on investment, ROI) по мере добавления ресурсов для обработки нагрузки и увеличения мощности. Предположим, у нас есть система с одним сервером и мы можем измерить ее максимальную мощность. Этот сценарий проиллюстрирован на рис. 11.1.

Теперь добавим еще один сервер, и мощность системы удвоится (рис. 11.2).

Это линейная масштабируемость. Мы удвоили количество серверов и как результат удвоили мощность системы. Большинство систем не являются линейно масштабируемыми, они часто масштабируются так, как показано на рис. 11.3.

[1] Джастин Бибер, мы тебя по-прежнему любим!

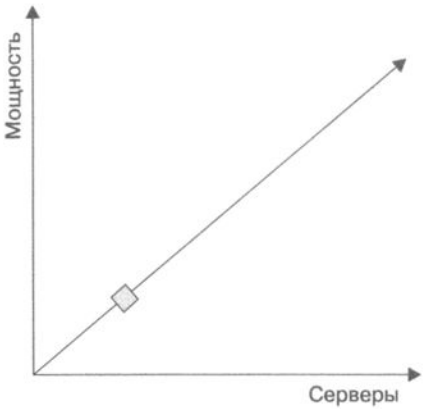


Рис. 11.1. Система с одним сервером

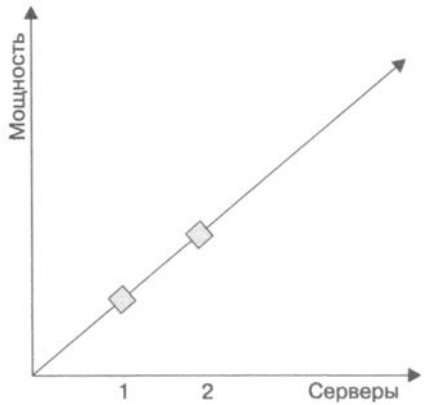


Рис. 11.2. У линейно масштабируемой системы при двух серверах двойная мощность

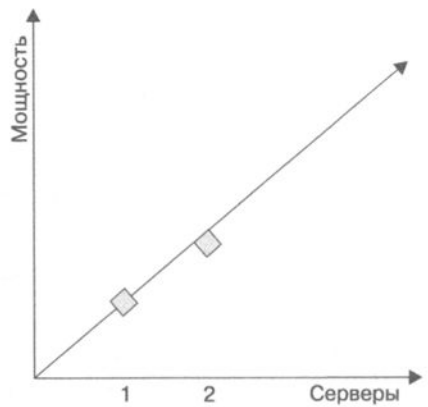


Рис. 11.3. Система, не масштабируемая линейно

Большинство систем при низких коэффициентах масштабирования обеспечивают масштабируемость чуть хуже линейной, однако отклонение от линейности становится более заметным при высоких коэффициентах масштабирования. Фактически большинство систем в конце концов достигают точки максимальной пропускной способности, за которой дополнительные инвестиции обеспечивают отрицательную отдачу, — добавьте рабочую нагрузку, и вы фактически уменьшите пропускную способность системы![1]

Почему это возможно? На протяжении многих лет с разной степенью успеха и реализма создавались различные модели масштабируемости. Модель масштабируемости, на которую ссылаемся мы, основана на некоторых базовых механизмах, которые начинают влиять на системы по мере их масштабирования. Это универсальный закон о масштабируемости доктора Нейла Дж. Гюнтера (Neil J. Gunter) (Universal Scalability Law, USL). Доктор Гюнтер подробно рассказал об этом в своих книгах, в том числе *Guerrilla Capacity Planning* (издательство Springer). Здесь мы не будем углубляться в математику, но если вам интересно, эта книга и учебные курсы, предлагаемые его компанией Performance Dynamics, могут оказаться полезными[2].

Краткое введение в USL заключается в том, что отклонение от линейной масштабируемости можно моделировать двумя факторами: одна часть работы не может выполняться параллельно, а другая требует перекрестных коммуникаций. Моделирование первого фактора приводит к известному закону Амдала, который гарантирует выравнивание пропускной способности. Когда часть задачи не может быть распараллелена независимо от того, как вы разделяете и властвуете, выполнение задачи потребует как минимум столько же времени, сколько требуется для реализации последовательных инструкций.

Добавление второго фактора — внутриузловой или внутрипроцессной коммуникации — к закону Амдала приводит к USL. Затраты на эту коммуникацию зависят от количества каналов связи, которое увеличивается квадратично относительно числа работников в системе. Таким образом, затраты в конечном счете растут быстрее, чем выгода, и именно из-за этого получается такая ретроградная масштабируемость. На рис. 11.4 изображены три рассмотренные концепции: линейное масштабирование, масштабирование по закону Амдала и масштабирование USL. Большинство реальных систем выглядят как кривая USL.

---

[1] Фактически термин «рентабельность инвестиций» можно рассматривать также в контексте ваших финансовых вложений. Модернизация компонента, позволяющая удвоить его мощность, часто требует вложений, более чем в два раза превышающих первоначальные инвестиции. Мы часто сталкивались с этим на практике, но здесь не будем рассматривать этот момент, чтобы не усложнять и так уже запутанную тему.

[2] Можете также прочитать нашу статью Forecasting MySQL Scalability with the Universal Scalability Law, в которой приводится краткий обзор математического аппарата и принципов работы в USL. Статья доступна по адресу http://www.percona.com.

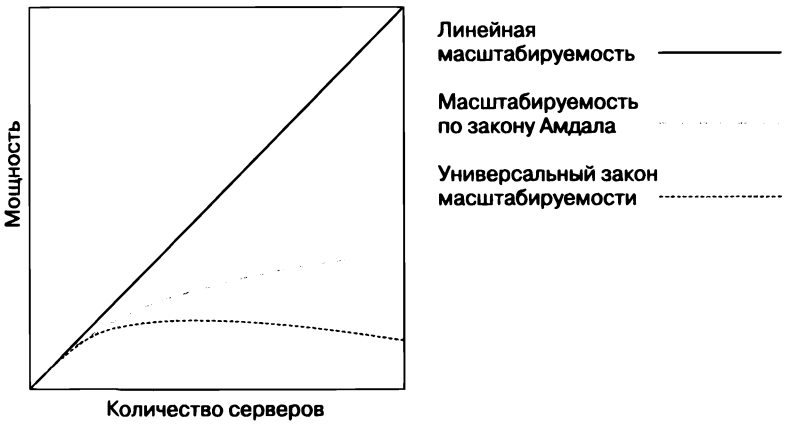


Рис. 11.4. Сравнение разных типов масштабируемости: линейной, по закону Амдала и по закону универсальной масштабируемости

USL можно применять как к аппаратным средствам, так и к программному обеспечению. В первом случае на оси *X* откладываются единицы оборудования, такие как серверы или процессоры, а рабочая нагрузка, объем данных и сложность запросов на единицу оборудования должны оставаться постоянными[1]. В случае программного обеспечения ось *X* на графике показывает конкурентные единицы, такие как пользователи или потоки, рабочая нагрузка на конкурентную единицу должна оставаться постоянной.

Важно понимать, что USL не сможет отлично описать какую-то реальную систему, поскольку это упрощенная модель. Тем не менее это хорошая основа для того, чтобы понять, почему системы по мере роста не обеспечивают постоянную отдачу от каждого вложенного рубля. Он также раскрывает важный принцип построения высокомасштабируемых систем: старайтесь избегать последовательных операций и перекрестных коммуникаций внутри системы.

Можно измерить систему и использовать регрессию для определения объема последовательных операций и перекрестных коммуникаций. Вы можете принять эти данные за наилучшую верхнюю границу для планирования мощности и прогнозирования производительности. А также изучить, насколько система отклоняется от модели USL, используя ее как наихудшую нижнюю границу для определения областей, в которых система не работает, как следовало бы. В обоих случаях USL даст вам основания для обсуждения масштабируемости. Без этого вы смотрели бы на систему и не понимали, чего от нее следует ожидать. Полное исследование этой темы заслуживает отдельной книги, и доктор Гюнтер уже написал ее, поэтому мы не будем вдаваться в подробности.

[1] На практике очень сложно точно определить аппаратную масштабируемость, поскольку трудно сохранить все эти параметры постоянными при изменении количества серверов в системе.

Понять проблемы масштабируемости может помочь теория ограничений, в которой объясняется, как повысить пропускную способность и эффективность системы за счет сокращения зависимых событий и статистических изменений. Она рассмотрена в книге Элиаху М. Голдратта (Eliyahu M. Goldratt) *The Goal* (издательство North River), которая представляет собой расширенную притчу о менеджере на производственном объекте. Хотя может показаться, что эта книга далека от области сервера базы данных, рассмотренные в ней принципы совпадают с принципами теории очередей и другими аспектами оперативных исследований.

Масштабируемые модели — это не последнее слово

Это все теория, но насколько хорошо она работает на практике? Точно так же, как законы Ньютона, которые оказались приближениями, довольно хорошо работающими, когда вы не близки к скорости света, законы масштабируемости — это упрощенные модели, которые хорошо работают в некоторых случаях. Иногда говорят, что все модели ошибочны. Тем не менее некоторые модели полезны, в частности, USL полезен для понимания некоторых факторов, влияющих на плохую масштабируемость.

USL перестает работать, когда возникают всякие тонкости во взаимодействии рабочей нагрузки с системой, на которой она выполняется. Например, особенно часто встречающаяся ситуация, которую USL не удается хорошо моделировать, такова: поведение системы меняется, поскольку общий размер памяти кластера изменяется относительно размера набора данных. USL не допускает возможности лучшего масштабирования, чем линейное, но в реальном мире мы иногда видим, как это происходит. Например, когда добавляем ресурсы в систему и заменяем нагрузку, использующую в том числе и операции ввода/вывода на диск, на нагрузку, работающую исключительно с памятью.

Существуют и другие случаи, когда модель USL не очень хорошо описывает поведение системы. Она не моделирует все возможные способы изменения алгоритмической сложности при изменении размера систем или изменении набора данных. (USL имеет компоненты O(1) и O(N), но как насчет компонента O(log N) или O(N log N), например?) Немного поразмыслив и попрактиковавшись, мы могли бы, вероятно, расширить USL и тем самым охватить некоторые из этих распространенных случаев. Однако это превратит простую и удобную модель в более запутанную и сложную для использования. Это довольно полезная модель, которая во многих случаях вполне применима на практике. Она хорошо моделирует обычное поведение системы, а справиться с отклонениями от модели ваш разум должен самостоятельно. Вот поэтому мы считаем, что эта модель — хороший компромисс между корректностью и применимостью. Короче говоря, отнеситесь к ней с изрядной долей скепсиса и, использовав ее, проверьте, оправдываются ли ваши ожидания.

Масштабирование MySQL

Размещение всех данных приложения на единственном экземпляре MySQL — не тот подход, который хорошо масштабируется. Рано или поздно увеличившаяся нагрузка на сервер приведет к появлению узких мест и, как следствие, к снижению производительности. Традиционно в этом случае просто покупают более мощные серверы. Такой подход называется вертикальным масштабированием. Альтернативный подход — распределение работы между несколькими компьютерами — называется горизонтальным масштабированием. Мы обсудим, как объединить горизонтальную и вертикальную масштабируемость при консолидации и как выполнять масштабирование с использованием кластерных решений. В большинстве приложений имеются данные, которые используются редко или не используются вовсе, их можно удалить или заархивировать. Такой подход мы называем обратным масштабированием, просто чтобы сохранить аналогию с другими терминами.

Планирование масштабируемости

О масштабируемости начинают задумываться, когда у сервера возникают трудности при увеличении нагрузки. Обычно это проявляется как снижение производительности — в виде замедления обработки запросов, смещения рабочей нагрузки от использования процессора в сторону ввода/вывода, роста конкуренции между запросами и увеличения задержки. Как правило, причина связана с увеличением сложности запросов или с тем, что какая-то часть данных или индекса, которая раньше помещалась в память, теперь в нее не помещается. Возможно, ухудшение будет заметно лишь для некоторых видов запросов, а другие продолжат работать, как раньше. Нередко длинные и сложные запросы начинают страдать раньше, чем более простые.

Если приложение масштабируемо, то достаточно добавить несколько серверов, и трудности с производительностью исчезнут. В противном случае вы будете постоянно с чем-то воевать. Этого можно избежать, если планировать масштабируемость с самого начала.

Самая трудная часть планирования масштабируемости — оценить, с какой нагрузкой придется иметь дело. Абсолютно точная оценка не нужна, но желательно знать хотя бы порядок величины. Если оценка завышена, то некоторое время будет зря потрачено на разработку, а если занижено, то возросшая нагрузка застанет вас врасплох.

Кроме того, следует приблизительно оценить темпы роста нагрузки, то есть знать, где находится горизонт. Для некоторых приложений простенький прототип вполне может работать в течение нескольких месяцев, а за это время вы сможете привлечь инвестиции и создать более масштабируемую архитектуру. Для других текущая архитектура должна обладать запасом мощности хотя бы на два года.

Планируя масштабируемость, вы должны задать себе следующие вопросы.

- Насколько полна функциональность приложения? Многие решения, которые мы собираемся предложить, усложняют реализацию некоторых возможностей. Если вы еще не реализовали какие-то базовые функции, то не исключено, что потом их

будет трудно встроить в масштабированное приложение. Аналогично не всегда легко принять правильное решение о масштабировании, не зная, как эти функции работают в действительности.

- Какова ожидаемая пиковая нагрузка? Приложение должно ее выдерживать. Что случится, если ваш сайт станет таким же популярным, как начальная страница Yahoo! News или Slashdot? Но даже если ваше приложение не является сверхпопулярным сайтом, все равно оно может испытывать пиковые нагрузки. Например, для сайта интернет-магазина пиковыми являются праздничные дни, особенно те, которые традиционно посвящают шопингу, например, несколько предрожденственских недель. В США День святого Валентина и уик-энд перед Днем матери часто становится пиком продаж в интернет-магазинах, торгующих цветами.
- Если требуется, чтобы нагрузку выдерживали все части системы, то что случится, если какая-то из них откажет? Например, если операции чтения должны быть распределены по реплицируемым подчиненным серверам, то сохранит ли система работоспособность, если один из них выйдет из строя? Не придется ли отключать часть функциональности? На случай возникновения подобных проблем можно заложить избыточную мощность.

Перед тем как приступать к масштабированию

Как хорошо было бы жить в идеальном мире, где все распланировано заранее, разработчиков достаточно, бюджетных ограничений не существует и т. д. Увы, в реальности все намного сложней, и при масштабировании приложения приходится идти на компромиссы. В частности, не исключено, что некоторые крупные изменения в проекте придется на время отложить. Прежде чем вплотную заняться рассмотрением масштабирования MySQL, приведем перечень того, что можно сделать сразу, еще не предпринимая серьезных усилий по масштабированию.

- *Оптимизировать производительность.* Часто существенного выигрыша можно добиться ценой сравнительно простых изменений, например, правильно проиндексировав таблицы или перейдя с подсистемы хранения MyISAM на InnoDB. Если производительность ограничена уже сейчас, то первым делом нужно включить и проанализировать журнал медленных запросов (см. главу 3).

  Существует точка сокращения отдачи. После того как бóльшая часть серьезных проблем исправлена, становится все сложнее улучшать производительность. Каждая следующая оптимизация требует все больших усилий и дает все менее заметный результат, хотя сложность приложения при этом быстро возрастает.
- *Приобрести более мощное оборудование.* Иногда модернизация имеющихся серверов или добавление новых дает неплохие результаты. Приобретение нескольких лишних серверов или памяти особенно целесообразно, когда приложение находится на ранней стадии жизненного цикла. В качестве альтернативы можно попробовать оставить приложение на одном сервере. Как правило, практичнее купить дополнительные единицы оборудования, чем менять архитектуру приложения. Особенно если время поджимает, а разработчиков не хватает.

Покупка дополнительного оборудования — приемлемое решение, если приложение либо невелико, либо спроектировано так, что может воспользоваться новым оборудованием. Это типичная ситуация для вновь созданных продуктов, которые еще не выросли или правильно спроектированы. Приобретение оборудования для зрелых крупномасштабных приложений может ничего не дать либо оказаться слишком дорогостоящим. Например, переход от одного сервера к трем обойдется дешево, а от 100 к 300 — совсем другое дело, дороговато. В таком случае имеет смысл направить время и усилия на выжимание всей возможной производительности из существующей системы.

Вертикальное масштабирование

Вертикальное масштабирование означает покупку более мощного оборудования, и для многих приложений это все, что вам нужно сделать. У этой стратегии есть множество преимуществ. Например, один сервер, в отличие от нескольких, гораздо проще поддерживать и развивать, что обеспечивает значительную экономию средств. Резервное копирование и восстановление приложения на одном сервере также выполняется проще, поскольку никогда не возникает вопроса о согласованности или о том, какой набор данных является правильным. Есть и другие причины. Затраты связаны со сложностью, а вертикальное масштабирование проще, чем горизонтальное.

Масштабировать по вертикали можно довольно далеко. Сегодня на рынке доступны серверы с объемом памяти 0,5 Тбайт, с 32 или большим количеством ядер процессора и большей мощностью ввода/вывода, чем вы можете использовать в MySQL (например, флеш-память на PCIe-картах). Благодаря умному проектированию приложений и баз данных и хорошим навыкам оптимизации производительности можете создавать очень большие приложения для работы с MySQL на таких серверах.

Насколько сильно можно масштабировать MySQL на современном оборудовании? Хотя ее можно запустить на очень мощных серверах, окажется, что, как и большинство серверов баз данных, MySQL масштабируется не великолепно (неожиданно!) при добавлении аппаратных ресурсов. Для запуска MySQL на больших железных ящиках вам обязательно понадобится последняя версия сервера. Из-за проблем с внутренней масштабируемостью версии MySQL 5.0 и 5.1 плохо работают на таком большом оборудовании. Вам понадобится MySQL 5.5 или более новых версий или Percona Server 5.1 или новее. Тем не менее в настоящее время разумная точка уменьшения отдачи примерно такова: 256 Гбайт ОЗУ, 32 ядра и флеш-накопитель PCIe. MySQL будет продолжать улучшать производительность на более крупном оборудовании, чем это, однако соотношение цены и производительности не будет таким же хорошим. На самом деле даже в этих системах вы часто можете получить гораздо лучшую производительность за счет запуска нескольких меньших экземпляров MySQL вместо одного большого, который использует все ресурсы сервера. Но все настолько быстро меняется, что этот совет, вероятно, быстро устареет.

Вертикальное масштабирование может работать в течение некоторого времени, но лишь до тех пор, пока масштаб приложения не превысит критическую отметку[1].

Первая причина этого — деньги. Вне зависимости от программного обеспечения в какой-то момент наращивание мощности оборудования становится неприемлемым из финансовых соображений. Существует некий диапазон аппаратных решений с оптимальным соотношением цены и производительности. За пределами этого диапазона находится в основном специализированное и, соответственно, более дорогое оборудование. Тем самым положен практический предел тому, как далеко можно зайти при вертикальном масштабировании. Даже если главный сервер способен эффективно использовать много процессоров, маловероятно, что вам удастся подобрать подчиненный сервер, который успевал бы реплицировать данные. Сильно загруженный главный сервер легко справляется с гораздо бóльшим объемом работы, чем подчиненный сервер с таким же оборудованием. Связано это с тем, что поток репликации на подчиненном сервере не может толком задействовать несколько процессоров и дисков.

Наконец, невозможно масштабировать по вертикали до бесконечности, поскольку даже самые мощные компьютеры имеют пределы. Обычно приложения, работающие на одном сервере, сначала упираются в ограничения при чтении, особенно при обработке сложных запросов выборки. Такие запросы внутри MySQL функционируют в однопоточном режиме, а следовательно, используют всего один ЦП, и ни за какие деньги вы не сможете существенно повысить их производительность. Самые быстрые из имеющихся на рынке процессоров, специально предназначенные для серверов, всего-то раза в два быстрее стандартных потребительских ЦП. Добавление нескольких процессоров или процессорных ядер не поможет медленным запросам работать быстрее. При этом по мере того, как данные перестают умещаться в кэше, сервер начинает ощущать нехватку памяти. Обычно это проявляется в виде интенсивного обращения к дискам, а диски — один из самых медленных компонентов современного компьютера.

Наиболее очевидное место, где невозможно вертикальное масштабирование, — это облако. Обычно вы не можете использовать очень мощные серверы в большинстве общественных облаков, поэтому, если ваше приложение должно вырасти очень большим, вертикальное масштабирование ситуацию не спасет. Мы обсудим эту тему в главе 13.

По этим причинам рекомендуем не планировать вертикальное масштабирование в тех случаях, когда вполне реально упереться в потолок масштабируемости, что станет серьезной проблемой для бизнеса. Если вы знаете, что приложение будет расти очень быстро, то вполне допустимо купить более мощный сервер на тот непродолжительный промежуток времени, пока вы будете работать над другим решением. Однако рано или поздно вам все же придется перейти к горизонтальному масштабированию, что и является темой следующего раздела.

---

[1] Мы избегаем слова «веб-масштабирование», поскольку оно потеряло всякий смысл (см.: http://www.xtranormal.com/watch/6995033/).

Масштабирование по горизонтали

Мы можем разбить тактику масштабирования на три большие группы: репликация, секционирование и шардирование.

Простейший и наиболее распространенный способ масштабирования по горизонтали — распределить данные по нескольким серверам с помощью репликации, а затем выполнять чтение на подчиненных серверах. Эта техника хорошо работает в приложениях, где много запросов на чтение. У нее есть недостатки, например дублирование кэша, но даже эта проблема может оказаться не слишком серьезной, если размер данных ограничен. Мы немало говорили на данную тему в предыдущей главе и еще вернемся к ней в текущей.

Еще один широко употребительный метод горизонтального масштабирования — секционирование рабочей нагрузки по нескольким узлам. Конкретный способ секционирования требует тщательного обдумывания. Большинство крупномасштабных приложений на основе MySQL не автоматизируют секционирование или по крайней мере автоматизируют не полностью. В этом разделе мы рассмотрим некоторые варианты секционирования и обсудим их сильные и слабые стороны.

Узлом называется функциональная единица в архитектуре MySQL. Если вы заранее не планируете избыточность и высокую доступность, то узлом может быть и один сервер. Если же проектируется система с резервированием и возможностью аварийного переключения при отказе, то узлом могут быть:

- пара серверов в топологии репликации «главный сервер — главный сервер» с активным главным и пассивным подчиненным сервером;
- один главный и много подчиненных серверов;
- активный сервер, использующий распределенное реплицируемое блочное устройство (DRBD) в качестве резерва;
- кластер на базе SAN.

В большинстве случаев на всех серверах внутри одного узла должны храниться одинаковые данные. Нам нравится схема с репликацией типа «главный сервер — главный сервер», состоящая из двух серверов в активном и пассивном режиме.

Функциональное секционирование

Функциональное секционирование, или разделение обязанностей, означает, что под разные задачи выделяются разные узлы. Мы уже встречались с несколькими подобными подходами — например, в предыдущей главе упоминалось о назначении разных серверов для OLTP- и OLAP-запросов. Функциональное секционирование развивает эту идею, предлагая назначать отдельные серверы или узлы разным приложениям так, чтобы в каждом узле хранились только данные, необходимые конкретному приложению.

Здесь мы употребляем слово «приложение» в широком смысле. Имеется в виду не отдельная программа, а набор взаимосвязанных программ, которые можно легко отделить от другого ПО, не имеющего к ним отношения. Например, если сайт включает разделы, не связанные общими данными, то можно секционировать его по функциональному назначению. Нередко можно встретить порталы, на которых представлены не связанные между собой области: из портала можно перейти в новостной раздел, форумы, область технической поддержки, базы знаний и т. д. Данные для каждой из этих функциональных областей могут храниться на отдельном сервере MySQL. Такая организация представлена на рис. 11.5.

Рис. 11.5. Портал и узлы, выделенные под различные функциональные области

Если приложение очень велико, то под каждую функциональную область можно отвести свой веб-сервер, хотя так делают редко.

Еще один подход к функциональному секционированию заключается в том, чтобы разделить данные одного приложения на множество таблиц, которые никогда не соединяются друг с другом. При необходимости можно иногда соединять таблицы из разных множеств либо на прикладном уровне, если это не критично для производительности. У этого приема есть несколько вариаций, но общим для них является то, что данные каждого типа находятся на одном узле. Такой способ секционирования применяется нечасто, поскольку его очень трудно реализовать эффективно, а существенных преимуществ по сравнению с другими методами он не дает.

Однако и возможности функционального секционирования небезграничны, так как если некоторая область связана с одним узлом MySQL, то она должна масштабироваться по вертикали. Рано или поздно какое-нибудь приложение или функциональная область разрастется слишком сильно, и тогда придется искать другую стратегию. А если зайти по этому пути слишком далеко, то впоследствии изменить архитектуру на лучше масштабируемую может оказаться сложно.

Шардирование данных

Шардирование данных[1] сейчас является наиболее распространенным и успешным подходом к масштабированию очень больших приложений на базе MySQL. Для этого данные разбивают на меньшие куски, или секции (shards), и хранят их на разных узлах.

Шардирование хорошо сочетается с некоторыми видами функционального секционирования. В большинстве подобных систем есть также некоторые глобальные значения, которые не шардируются вовсе (скажем, списки городов или логины). Обычно такие данные хранятся на одном узле, доступ к которому часто организуется через специальный кэш, например memcached.

На практике в большинстве приложений шардируется только информация, которая в этом нуждается, — как правило, те части набора данных, которые будут расти особенно быстро. Предположим, что вы создаете службу блогов. Если ожидается 10 миллионов пользователей, то, быть может, шардировать регистрационную информацию пользователей и не обязательно, так как вся она (или по крайней мере относящаяся к активным пользователям) помещается в памяти. Если же ожидается 500 миллионов пользователей, то шардировать их профили имеет смысл. Генерируемый пользователями контент — сообщения и комментарии — почти наверняка придется шардировать в любом случае, поскольку эти записи гораздо длиннее и их намного больше.

В больших приложениях может быть несколько логических наборов данных, которые можно шардировать по-разному. Их можно хранить на разных серверах, хотя это и не обязательно. Кроме того, одни и те же данные можно шардировать разными способами в зависимости от того, как организован доступ к ним. Пример такого подхода будет приведен в дальнейшем.

Проектирование приложений, построенных на основе шардирования данных, принципиально отличается от того, как обычно строят приложения на начальном этапе. Переход от монолитного хранилища данных к шардированной архитектуре может оказаться весьма непростым делом. Поэтому гораздо проще с самого начала закладывать идеологию шардирования, если есть основания полагать, что это понадобится в будущем.

Большинство приложений, в которые шардирование не было встроено изначально, впоследствии, по мере роста, проходят через промежуточные этапы. Например, в службе блогов можно поначалу воспользоваться репликацией для масштабирования запросов на чтение, но в конце концов это решение перестанет работать. Тогда ничто не помешает разбить службу на три части: пользователи, сообщения и комментарии. Их можно поместить на разные серверы (функциональное секционирование), возможно, с сервис-ориентированной архитектурой, а соединение таблиц выполнять на уровне приложения. На рис. 11.6 показан постепенный переход от использования одного сервера к функциональному секционированию.

---

[1] Употребляются также термины splintering (дробление) и partitioning (секционирование), но мы используем термин sharding (шардирование) во избежание путаницы. Google называет эту операцию шардированием. А если это подходит Google, это подойдет и нам.

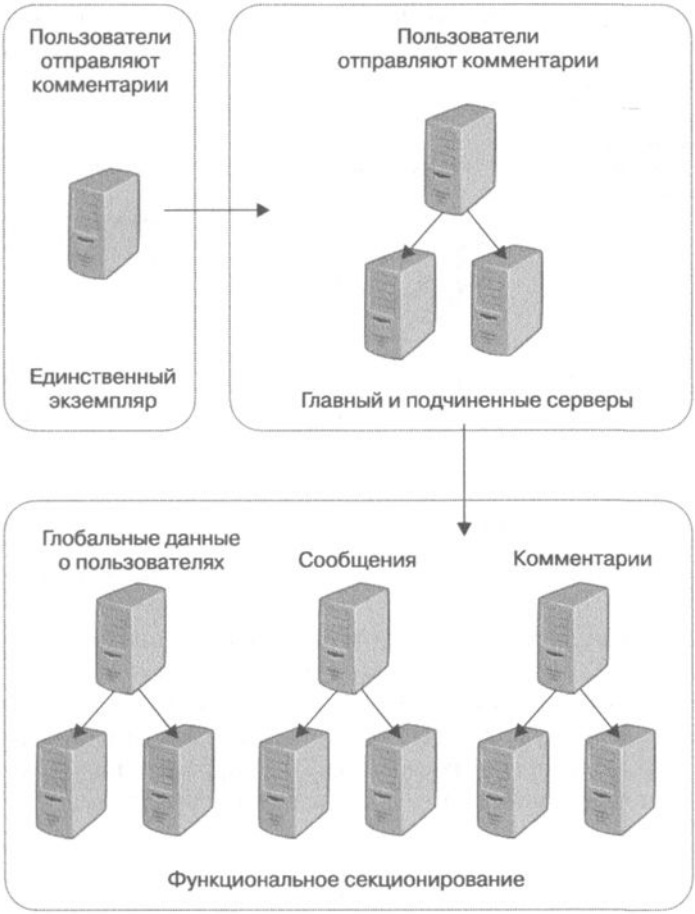


Рис. 11.6. От одного экземпляра к функционально секционированному хранилищу данных

Наконец, сообщения и комментарии можно шардировать по идентификатору пользователя, а информацию о пользователях хранить на отдельном узле. Если применить конфигурацию «главный — подчиненный» для глобального узла и пары «главный сервер — главный сервер» для шардированных узлов, окончательное хранилище данных может выглядеть так, как показано на рис. 11.7.

Если заранее известно, что потребуется масштабировать очень большую систему, и понятны ограничения функционального секционирования, то можно пропустить промежуточные этапы и сразу перейти от единственного узла к шардированному хранилищу. Дальновидность часто помогает избежать появления уродливых шардированных схем, что может произойти при постепенном решении возникающих задач.

В шардированных приложениях часто применяется библиотека абстрагирования базы данных, которая упрощает взаимодействие между программой и шардированным хранилищем. Обычно такие библиотеки не скрывают шардирование полностью, поскольку часто приложение знает о запросе нечто такое, что неизвестно хранилищу

данных. При слишком высоком уровне абстракции могут выполняться неэффективные действия, например опрос всех узлов в поисках информации, которая заведомо хранится только на одном узле.

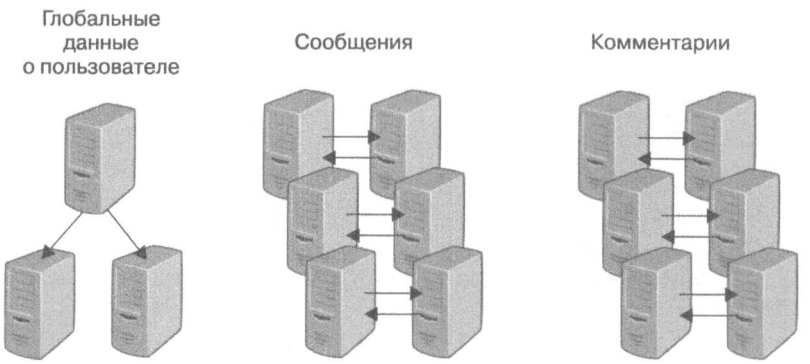


Рис. 11.7. Хранилище данных с одним глобальным узлом и шестью парами «главный — главный»

Шардированное хранилище данных может показаться элегантным решением, но его трудно реализовать. Так почему же мы голосуем за такую архитектуру? Ответ прост: если вы хотите масштабировать операции записи, то просто обязаны секционировать данные. Имея единственный главный сервер, невозможно масштабировать запись, сколько бы ни было подчиненных серверов. Шардирование при всех его недостатках — предпочтительное, на наш взгляд, решение этой проблемы.

Шардировать или не шардировать?

Вот в чем вопрос, не так ли? А вот простой ответ: не шардировать, если в этом нет необходимости. Посмотрите, можете ли вы избежать его с помощью оптимизации производительности, или улучшения приложения, или проектирования базы данных. Если шардирование можно отложить на довольно длительный срок, возможно, проще купить большой сервер, обновить MySQL до новой версии с более высокой производительностью и продолжать работать с одним сервером плюс-минус репликация.

Проще говоря, шардирование неизбежно, когда размер данных или рабочая нагрузка по операциям записи становятся слишком большими для одного сервера. Вы бы удивились, узнав, как сильно можно масштабировать системы, не прибегая к шардированию, — просто грамотно проектируя приложение. Некоторые очень популярные приложения, которые, как вы могли бы предположить, шардированы с самого начала, на самом деле выросли до продуктов с многомиллиардной стоимостью и безумным объемом трафика без шардирования. Это не единственная возможность. Это сложный способ создать приложение, и если он не нужен, не стоит его использовать.

Выбор ключа секционирования

Самая важная и сложная задача, возникающая при шардировании, — поиск и выборка данных. Способ поиска зависит от того, как они шардированы. Сделать это можно разными методами — одни получше, другие похуже.

Наша цель состоит в том, чтобы самые важные и часто встречающиеся запросы затрагивали как можно меньше шардов (помните, что один из принципов масштабируемости состоит в уменьшении перекрестных коммуникаций между узлами). Здесь очень важно выбрать ключ (или ключи) секционирования данных. Ключ секционирования определяет, в какой шард попадет та или иная строка. Если известен ключ секционирования некоторого объекта, то можно ответить на два вопроса.

- Где следует сохранить данные?
- Где найти запрошенные данные?

Далее мы покажем разные способы выбора и использования ключа секционирования. А пока рассмотрим пример. Допустим, что мы поступаем так же, как NDB Cluster, и применяем хеш первичного ключа таблицы для распределения данных по шардам. Это очень простой подход, но он плохо масштабируется, поскольку часто заставляет искать информацию во всех шардах. Например, если вы хотите найти все сообщения в блоге пользователя 3, где их искать? Вероятнее всего, они равномерно распределены по всем шардам, так как секционирование производилось по первичному ключу, а не по идентификатору пользователя. Использование хеша первичного ключа упрощает получение информации о том, где хранятся данные, но может затруднить их извлечение. Это зависит от того, какие данные вам нужны и знаете ли вы первичный ключ.

Запросы к нескольким шардам хуже, чем запросы к одному из них, но если они затрагивают не слишком много шардов, то все не так уж плохо. Самый худший случай, когда у вас нет ни малейшего представления о том, где находятся данные, и приходится просматривать все шарды без исключения.

Обычно хорошим ключом секционирования является идентификатор какой-нибудь важной сущности в базе данных. Такие сущности называются *единицами шардирования*. Например, если информация секционируется по идентификатору пользователя или клиента, то единицей шардирования является соответственно пользователь или клиент.

Для начала неплохо изобразить модель данных в виде диаграммы «сущность — связь» или эквивалентного представления, на котором видны все сущности и связи между ними. Постарайтесь нарисовать диаграмму так, чтобы взаимосвязанные сущности располагались близко друг к другу. Часто с помощью визуального изучения диаграммы можно найти таких кандидатов на роль ключа секционирования, которых иначе вы бы не заметили. Но не ограничивайтесь одной лишь диаграммой, примите во внимание также запросы, выполняемые приложением. Даже если между двумя сущностями существует какая-то связь, но соединение по ней производится

редко или вообще никогда, то при шардировании эту связь можно не принимать во внимание.

Одни модели данных лучше поддаются шардированию, чем другие, — все зависит от степени связности графа «сущность — связь». На рис. 11.8 слева изображена модель, которая легко шардируется, а справа — шардируемая с трудом.

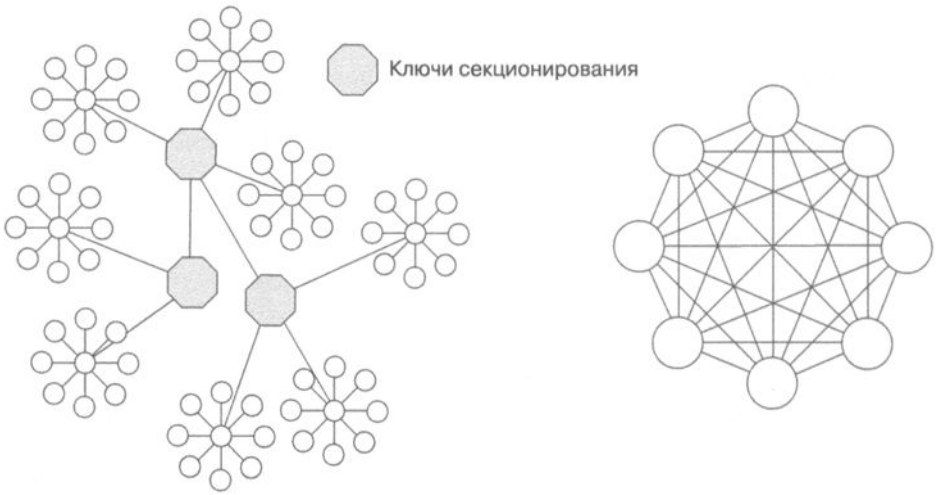


Рис. 11.8. Две модели данных — одна шардируется легко, другая с трудом

Левую модель данных легко шардировать, потому что в ней много связных подграфов, состоящих в основном из узлов, между которыми есть всего одна связь, и можно сравнительно легко разрезать связи между подграфами. Правую модель трудно шардировать, потому что в ней таких подграфов нет. Большинство моделей данных представляются скорее левой, чем правой диаграммой.

Выбирая ключ секционирования, попробуйте подобрать что-то, что позволит вам по максимуму избежать запросов к нескольким шардам, но чтобы при этом шарды были настолько маленькими, что у вас не будет проблем с непропорционально большими фрагментами данных. Желательно, чтобы шарды были одинаково маленькими, если это возможно, а если нет, то по крайней мере достаточно небольшими, чтобы их можно было легко сбалансировать, объединив в группы разное количество шардов. Например, если ваше приложение использует данные только о США и вы хотите разделить набор данных на 20 шардов, не стоит, по-видимому, шардировать на основе отдельных штатов, поскольку в Калифорнии очень большое население. Однако вы можете выполнить шардирование по графствам или телефонному коду местности, поскольку, хотя шарды и не будут равномерно наполнены, их окажется довольно много, чтобы вы могли сформировать 20 наборов, в совокупности наполненных примерно одинаково. Тогда вы сможете выбрать их так, чтобы избежать запросов к разным шардам.

Несколько ключей секционирования

Если модель данных сложна, то и шардировать ее труднее. Во многих приложениях существует более одного ключа секционирования, особенно если в данных можно выделить несколько важных параметров. Иными словами, приложение должно иметь возможность эффективно взглянуть на информацию под разными углами зрения и получить целостное представление. Это означает, что некоторые данные придется хранить в двух разных видах.

Например, данные приложения для ведения блогов можно шардировать по идентификатору пользователя и идентификатору сообщения, поскольку то и другое — универсальные способы взглянуть на данные. Действительно, часто вы хотите видеть все сообщения одного пользователя и все комментарии к сообщению. Но шардирование по пользователям не поможет в поиске комментариев к сообщению, а шардирование по сообщениям бесполезно для поиска сообщений данного пользователя. Если вы хотите, чтобы оба запроса затрагивали только один шард, то придется шардировать обоими способами.

Тот факт, что необходимы несколько ключей секционирования, еще не означает, что требуется проектировать два дублирующих друг друга хранилища данных. Рассмотрим еще один пример: сайт социальной сети книголюбов, на котором пользователи могут оставлять свои комментарии о книгах. На нем могут отображаться все комментарии к некоторой книге, а также все книги, которые прочитал и прокомментировал данный пользователь.

В таком случае можно построить одно шардированное хранилище с данными о пользователях и второе — с данными о книгах. В комментариях хранится как идентификатор пользователя, так и идентификатор книги, поэтому они пересекают границы шардов. Вместо того чтобы полностью дублировать комментарии, можно хранить их вместе с данными о пользователях. Тогда вместе с данными о книгах достаточно хранить только заголовок и идентификатор комментария. При этом для построения большинства страниц, содержащих комментарии к книгам, не придется обращаться к обоим хранилищам данных, ну а если потребуется вывести полный текст комментария, то его можно будет достать из хранилища данных о пользователях.

Запросы в разных шардах

В большинстве шардированных приложений встречаются запросы, требующие агрегирования или соединения данных, хранящихся в разных шардах. Например, если на сайте книголюбов нужно показать наиболее активных или наиболее популярных пользователей, то по определению необходимо обращаться ко всем шардам. Оптимизация подобных запросов — самая сложная часть реализации шардирования, поскольку то, что приложение видит как один запрос, на самом деле надо разделить и параллельно выполнить в каждом шарде. Хорошо написанный слой абстрагирования базы данных может в какой-то мере облегчить решение этой задачи, но все равно такие

запросы выполняются гораздо медленнее запросов к одному шарду, к тому же настолько затратны, что обычно необходимо применять агрессивное кэширование.

Некоторые языки, в частности PHP, плохо поддерживают параллельное выполнение нескольких запросов. Чтобы обойти эту трудность, обычно разрабатывают вспомогательное приложение, часто на языке C или Java, которое отправляет запросы и объединяет результаты. Тогда PHP-страница может обратиться к этому приложению, обычно представляющему собой веб-сервис или рабочий сервис, такой как Gearman.

Выполнение запросов к нескольким шардам можно ускорить за счет наличия сводных таблиц. Для их заполнения нужно обойти все шарды и по завершении сохранить данные также на серверах каждого шарда, чтобы обеспечить резервирование. Если дублирование информации на каждом шарде слишком расточительно, то можно собрать все сводные таблицы в каком-нибудь специальном хранилище.

Нешардированные данные часто размещаются на глобальном узле, а для его защиты от перегрузки активно применяют кэширование.

В некоторых приложениях используется случайное шардирование в тех случаях, когда важно равномерное распределение данных или подходящего ключа секционирования не существует. Хорошим примером могут служить распределенные программы поиска, где запросы в разных шардах и агрегирование являются нормой, а не исключением.

Запросы в разных шардах — не единственная сложность, встречающаяся при организации шардирования. Трудно также поддерживать согласованность данных. Внешние ключи через границы шардов не действуют, поэтому обычно контроль производится на уровне приложения. Можно использовать внешние ключи внутри шарда, поскольку согласованность данных внутри него является самой важной задачей. Можно обратиться к XA-транзакциям, но из-за высоких издержек так поступают редко.

Можно также спроектировать процесс очистки, который время от времени будет запускаться. Например, если срок хранения учетной записи в клубе книголюбов истек, то необязательно удалять ее немедленно. Можно написать периодическое задание, которое удалит комментарии пользователя из шарда с данными о книгах. Можно также подготовить скрипт, который станет периодически запускаться и проверять согласованность информации в разных шардах.

Распределение данных по шардам и узлам

Между шардами и узлами не обязательно должно существовать взаимно однозначное соответствие. Часто бывает разумнее сделать размер шарда значительно меньше, чем емкость узла, чтобы в одном узле можно было хранить несколько шардов.

Чем меньше размер шарда, тем проще обслуживать находящиеся в нем данные. В результате упрощаются резервное копирование и восстановление информации, а кроме того, для небольших таблиц быстрее завершаются такие действия, как изменение схемы. Пусть, например, имеется таблица размером 100 Гбайт. Ее можно

хранить целиком в одном месте или разбить на 100 шардов по 1 Гбайт, разместив их все на одном узле. Теперь предположим, что над этой таблицей нужно построить новый индекс. На 100-гигабайтном шарде это заняло бы гораздо больше времени, чем требуется на совокупное индексирование гигабайтных кусочков, поскольку шард размером 1 Гбайт целиком помещается в память. Возможно, вам потребуется во время работы команды `ALTER TABLE` сделать данные недоступными, тогда лучше заблокировать 1 Гбайт, а не все 100.

Также шарды меньшего размера удобнее перемещать, что упрощает задачу перераспределения емкости и поиска оптимального баланса между шардами и узлами. Вообще говоря, перемещение шарда не слишком эффективная процедура. Обычно нужно перевести его в режим только для чтения (эту возможность необходимо встроить в приложение), выгрузить данные и переместить их на другой узел. Для выгрузки чаще всего используется программа mysqldump, а для загрузки — mysql. Если вы работаете с Percona Server, то можете применить XtraBackup для перемещения файлов между серверами, что намного эффективнее, чем выгрузка данных и их перезагрузка.

Помимо перемещения шардов между узлами, нужно подумать еще и о перемещении данных между шардами, желательно не прекращая обслуживания пользователей. Если шарды слишком велики, то балансировать мощность путем перемещения шардов целиком будет труднее, так что придется придумать способ переноса отдельных логических блоков данных (например, данных об одном пользователе) из одного шарда в другой. Перемещение данных между шардами обычно намного сложнее, чем перемещение самих шардов, поэтому по возможности старайтесь этого избегать. Вот почему мы рекомендуем ограничивать размер шарда так, чтобы им можно было управлять.

Относительные размеры шардов зависят от нужд приложения. Определяя, что такое поддающийся управлению размер, мы обычно предлагаем такую грубую оценку: это размер, при котором таблицы настолько малы, что для регулярного выполнения таких задач обслуживания, как `ALTER TABLE`, `CHECK TABLE`, `OPTIMIZE TABLE`, достаточно 5–10 минут.

Если размер шардов слишком мал, то может появиться чересчур много таблиц, что негативно отразится на файловой системе или на внутренних структурах MySQL. К тому же чем меньше размер шардов, тем больше запросов придется выполнять в разных шардах.

Организация шардов на узлах

Необходимо решить, как организовывать шарды на одном узле. Есть несколько распространенных методов.

- Размещать в каждом шарде одну базу данных и называть все эти базы одинаково. Такой метод обычно применяют, когда хотят, чтобы каждый шард повторял структуру исходного приложения. Он неплохо работает, если запускаются несколько экземпляров приложения, каждый из которых знает только об одном шарде.

- Помещать таблицы из нескольких шардов в одну базу данных и включать номер шарда в имена таблиц (например, `bookclub.comments_23`). При такой конфигурации в одной базе данных могут храниться несколько шардов.
- Использовать одну базу данных на каждый шард и помещать в эту базу все таблицы приложения. Включать номер шарда в имя базы, а не таблицы (то есть полное имя таблицы будет имеет вид `bookclub_23.comments`, `bookclub_23.users` и т. д.). Этот метод применяется, когда приложение соединяется с одной базой данных, а имя базы в запросах не указывается. Преимущество состоит в том, что запросы не нужно подстраивать под каждый шард. За счет этого упрощается переход к архитектуре шардов для приложений, работающих только с одной базой.
- Использовать одну базу данных на каждый шард и включать номер шарда как в имя базы, так и в имена таблиц (то есть полное имя таблицы имеет вид `bookclub_23.comments_23`).
- Запускать несколько экземпляров MySQL для каждого узла, каждый из которых есть один или несколько шардов, расположенных в любой разумной комбинации только что упомянутых способов.

Если номер шарда является частью имени таблицы, то нужен какой-то способ подставить его в шаблон запроса. Как правило, в шаблон включают специальные значения заполнителя в запросах, например спецификаторы в духе функции `sprintf` (скажем, `%s`) или строковую интерполяцию с помощью переменных. Вот один из способов записать шаблон запроса в PHP:

```
$sql = "SELECT book_id, book_title FROM bookclub_%d.comments_%d... ";
$res = mysql_query(sprintf($sql, $shardno, $shardno), $conn);
```

А вот как можно применить строковую интерполяцию:

```
$sql = "SELECT book_id, book_title FROM bookclub_$shardno.comments_$shardno ...";
$res = mysql_query($sql, $conn);
```

Эта идея легко реализуется в новых приложениях, но добавить шаблоны в существующую программу сложнее. При создании нового приложения, когда с шаблонами запросов проблем не возникает, мы предпочитаем заводить по одной базе на шард, включая номер шарда в имя базы и имена таблиц. Хотя такое решение несколько усложняет задачи наподобие включения команды `ALTER TABLE` в скрипты, у него есть и преимущества.

- Если шард целиком находится в одной базе данных, то его легко переместить с помощью утилиты mysqldump.
- Поскольку база данных представляет собой каталог в файловой системе, легко манипулировать файлами шарда.
- Если один шард не перемешан с другими, то его размер легко определяется.
- Глобально уникальные имена таблиц помогают избежать ошибок. Если имена таблиц, находящихся в разных местах, совпадают, то можно случайно обратиться не к тому шарду, установив соединение не с тем узлом, с которым нужно, или импортировать данные, предназначенные одному шарду, в таблицу, принадлежащую другому.

Имеет смысл задаться вопросом, не характерна ли для данных приложения близость шардов (shard affinity). Возможно, удастся получить выигрыш, поместив некоторые шарды недалеко друг от друга (на одном сервере, в одной подсети, в одном центре обработки данных или на узлах сети, обслуживаемых одним коммутатором), чтобы воспользоваться сходством типичных способов доступа к данным. Например, можно шардировать информацию по пользователям и поместить всех пользователей из одной страны в шарды, находящиеся на одних и тех же узлах.

Когда поддержка шардов добавляется в существующее приложение, часто предпочитают размещать по одному шарду на каждом узле. При таком упрощенном подходе минимизируется объем изменений в коде запросов, выполняемых приложением. С введением шардирования приложение и так придется серьезно переработать, а лишние проблемы никому не нужны. Если шардирование производится так, что на каждом узле оказывается уменьшенная копия всех данных приложения, то большую часть запросов изменять не придется, как не придется и задумываться о том, каким образом маршрутизировать запросы нужному узлу.

Фиксированное распределение

Существует два основных способа распределения данных по шардам: фиксированное и динамическое. Для обеих стратегий необходима функция секционирования, которая принимает на входе ключ секционирования строки и возвращает номер секции, в которой эта строка находится[1].

Для фиксированного распределения применяется функция разбиения, которая зависит только от значения ключа секционирования. В качестве примеров можно привести деление по модулю или хеш-функции. Такие функции отображают значения ключей секционирования на конечное число ячеек (buckets), в которых хранятся данные.

Пусть имеется 100 ячеек и требуется найти, в какую из них поместить пользователя `111`. Применение деления по модулю дает ответ: остаток от деления `111` на `100` равен `11`, поэтому пользователь должен находиться в шарде `11`.

Если же для хеширования задействована функция `CRC32()`, то в результате получается `81`:

```
mysql> SELECT CRC32(111) % 100;
+------------------+
| CRC32(111) % 100 |
+------------------+
|               81 |
+------------------+
```

[1] Здесь слово «функция» употребляется в математическом смысле как отображение множества входных значений (области определения) на множество выходных значений (область значений). Как будет показано, такую функцию можно реализовать разными способами, в том числе с использованием справочной таблицы в базе данных.

Основные достоинства фиксированного распределения — простота и низкие накладные затраты. К тому же ее легко встроить в приложение.

Однако у этой стратегии есть и недостатки.

- Если шарды велики и их немного, то сбалансировать нагрузку в них будет сложно.
- При фиксированном распределении вы лишены возможности решать, куда помещать конкретную запись, а это важно в приложениях, где нагрузка на единицы шардирования неравномерна. Некоторые части данных используются гораздо чаще, чем остальные, и когда такие обращения по большей части адресуются одному и тому же шарду, стратегия фиксированного распределения не позволяет снять нагрузку, переместив часть данных в другой шард. (Эта проблема не слишком серьезна, если элементы данных малы, но их количество в каждом шарде велико — закон больших чисел все расставит по своим местам.)
- Как правило, изменить алгоритм шардирования сложнее, потому что требуется перераспределить все существующие данные. Например, если шардирование производилось делением по модулю 10, то имеется 10 шардов. Когда приложение вырастет и шарды станут слишком большими, возникнет желание увеличить их количество до 20. Но для этого придется хешировать все заново, обновить очень много данных и перераспределить их по новым шардам.

В силу указанных ограничений мы обычно предпочитаем в новых приложениях пользоваться динамическим распределением. Но при добавлении шардирования в существующее приложение бывает проще применить фиксированную стратегию, поскольку она проще. Тем не менее большинство приложений, использующих фиксированное распределение, рано или поздно переходят к стратегии динамического распределения.

Динамическое распределение

Альтернативой фиксированному распределению является динамическое распределение, описание которого хранится отдельно в виде отображения единицы данных на номер шарда. Примером может служить таблица с двумя столбцами, в которых размещены идентификатор пользователя и идентификатор шарда:

```
CREATE TABLE user_to_shard (
    user_id INT NOT NULL,
    shard_id INT NOT NULL,
    PRIMARY KEY (user_id)
);
```

Функцией секционирования служит сама таблица. Зная ключ секционирования (идентификатор пользователя), можно найти соответствующий номер шарда. Если подходящей строки не существует, можно выбрать нужный шард и добавить строку в таблицу. Впоследствии сопоставление можно будет изменить, поэтому стратегия и называется динамической.

С динамическим распределением связаны определенные накладные затраты, так как требуется обращение к внешнему ресурсу, например серверу каталогов (узлу, на котором хранится отображение). Для поддержания приемлемой эффективности такая

архитектура часто нуждается в дополнительных слоях программного обеспечения. Например, можно использовать распределенную систему кэширования, в памяти которой хранятся данные с сервера каталогов, на практике изменяющиеся довольно редко. Или, что, вероятно, происходит чаще, вы можете просто добавить столбец `shard_id` в таблицу `users` и хранить его там.

Основное достоинство динамического распределения — более точное управление местом хранения данных. Это упрощает равномерное разделение данных по шардам и позволяет гибко адаптироваться к непредвиденным изменениям.

Кроме того, динамическое распределение дает возможность выстраивать многоуровневую стратегию шардирования поверх простого отображения ключей на шарды. Например, можно организовать двойное отображение, при котором каждой единице шардирования сопоставляется некоторая группа (например, группа членов клуба книголюбов), а сами группы по возможности размещаются в одном шарде. Это позволяет воспользоваться преимуществами близости шардов и избегать запросов между шардами.

Динамическое распределение дает возможность создавать несбалансированные шарды. Это полезно, когда не все серверы одинаково мощные или когда некоторые из них используются и для других целей, например архивирования данных. Если при этом имеется еще и возможность в любой момент перебалансировать шарды, то можно поддерживать взаимно однозначное соответствие между шардами и узлами, не растрачивая впустую емкость дисков. Некоторым нравится та простота, которая свойственна хранению в каждом узле ровно одного шарда (но не забывайте, что имеет смысл делать секции относительно небольшими).

Динамическое распределение и толковое применение близости шардов может предотвратить рост числа запросов между шардами по мере масштабирования. Представьте себе запрос к другому шарду в хранилище данных с четырьмя узлами. При фиксированном распределении может случиться так, что любой запрос затрагивает все шарды, тогда как динамическое распределение позволит сделать возможным запуск всего на трех узлах. Кажется, что разница невелика, но подумайте, что произойдет, когда количество шардов возрастет до 400: фиксированное распределение заставит опрашивать каждый из 400 шардов, а динамическое — все те же три.

Динамическое распределение позволяет организовать стратегию шардирования любой сложности, фиксированное такого богатства выбора не предоставляет.

Комбинирование динамического и фиксированного распределения

Можно также применять комбинацию динамического и фиксированного распределения. Зачастую это полезно, а иногда даже необходимо. Динамическое распределение хорошо работает, когда отображение не слишком велико. С ростом количества единиц шардирования его эффективность падает.

В качестве примера рассмотрим систему, в которой хранятся ссылки между сайтами. В ней необходимо хранить десятки миллиардов строк, а ключом секционирования

является комбинация исходного и конечного URL. (На любой из двух URL могут быть сделаны сотни миллионов ссылок, поэтому ни один из них по отдельности не является достаточно селективным.) Однако невозможно хранить все комбинации исходного и конечного URL в таблице сопоставления, поскольку их слишком много, а каждый URL занимает много места.

Одно из возможных решений — конкатенировать URL и написать хеш-функцию, отображающую получившиеся строки на фиксированное число ячеек, которые затем можно динамически отображать на шарды. Если количество ячеек велико, скажем 1 миллион, то в каждый шард их можно поместить довольно много. В результате мы сможем пользоваться преимуществами динамического шардирования, не заводя гигантскую таблицу отображения.

Явное распределение

Третья стратегия позволяет приложению явно выбрать шард при создании строки. Это трудно сделать для уже существующих данных, поэтому такое решение редко применяется при переработке имеющегося приложения с учетом шардирования. Однако иногда оно бывает полезно.

Смысл его заключается в том, чтобы закодировать номер шарда в идентификаторе: похожая техника применяется, чтобы избежать дублирования значений ключей в репликации типа «главный — главный» (см. подраздел «Запись на обоих главных серверах в конфигурации "главный — главный"» в главе 10).

Пусть, например, приложение хочет создать пользователя `3` и назначить ему шард `11` и для номера шарда вы отвели 8 старших бит в столбце типа `BIGINT`. Тогда пользователю будет присвоен идентификатор `(11 << 56) + 3`, или `792633534417207299`. Впоследствии приложение без труда сможет извлечь из этого числа и номер пользователя, и номер шарда. Делается это следующим образом:

```
mysql> SELECT (792633534417207299 >> 56) AS shard_id,
    -> 792633534417207299 & ~(11 << 56) AS user_id;
+----------+---------+
| shard_id | user_id |
+----------+---------+
|       11 |       3 |
+----------+---------+
```

Теперь предположим, что требуется создать для этого пользователя комментарий и сохранить его в том же самом шарде. Приложение может присвоить комментарию номер `5` и точно так же, как ранее, объединить его с номером секции — `11`.

Достоинство такого подхода заключается в том, что идентификатор каждого объекта уже несет в себе собственный ключ секционирования, тогда как остальные решения подразумевают соединение таблиц или иную операцию поиска для нахождения такого ключа. Если вы хотите извлечь из базы конкретный комментарий, то вовсе не обязательно знать, какому пользователю он принадлежит: сам идентификатор объекта

скажет, где его искать. Если бы объекты динамически шардировались по идентификатору пользователя, то пришлось бы сначала найти владельца комментария, а затем запросить у сервера каталогов, в каком шарде этот пользователь находится.

Еще одно решение — хранить ключ секционирования вместе с объектом, но в разных столбцах. Например, вы никогда не обращаетесь просто к комментарию с номером 5, а всегда к комментарию 5, принадлежащему пользователю 3. Возможно, кому-то такой подход понравится больше, поскольку он не нарушает первую нормальную форму, однако дополнительные столбцы увеличивают накладные затраты, требуют написания лишнего кода и вообще вызывают всяческие неудобства. (Это тот случай, когда, на наш взгляд, хранение двух значений в одном столбце оправданно.)

Недостаток явного распределения кроется в том, что шардирование фиксировано и заново балансировать шарды становится сложнее. В то же время этот метод хорошо работает в сочетании с комбинацией фиксированного и динамического распределения. Вместо того чтобы хешировать заранее заданное количество ячеек и затем отображать ячейки на узлы, ячейки просто кодируются в самом объекте. Это позволяет приложению управлять местонахождением данных и, в частности, помещать взаимосвязанные значения в один и тот же шард.

На сайте BoardReader (http://boardreader.com) применяется вариант этой методики: ключ секционирования закодирован в идентификаторе документа в системе Sphinx. Поэтому становится просто найти данные, ассоциированные с любым из результатов поиска, в шардированном хранилище данных. Дополнительную информацию о системе Sphinx см. в приложении А.

Мы включили описание комбинированного распределения, поскольку сталкивались со случаями, когда оно было полезно, но, вообще говоря, не рекомендуем применять его. Предпочитаем использовать динамическое распределение и по возможности избегать явного.

Повторное балансирование секций

При необходимости можно переместить данные из одного шарда в другой, чтобы сбалансировать нагрузку. Многие читатели, наверное, слышали, как разработчики больших сайтов фотогалерей или популярных социальных сетей рассказывали о своих инструментах перемещения пользователей в другие шарды.

Способность перемещать данные между шардами открывает целый ряд возможностей. Например, чтобы модернизировать оборудование, можно перенести всех пользователей из старого шарда в новый, не останавливая шард целиком и не переводя его в режим «только для чтения».

Однако мы стараемся по возможности избегать повторного балансирования, так как это может вызвать приостановку обслуживания. Из-за перемещения данных становится сложнее добавлять в приложение новые функции, поскольку их прихо-

дится учитывать в скриптах повторного балансирования. Если шарды не слишком велики, то прибегать к этому, возможно, и не понадобится: часто для балансирования нагрузки достаточно перенести шард целиком, а это гораздо проще, чем перемещать часть шарда (и более эффективно, если говорить о затратах в расчете на одну строку данных).

Одна из хорошо зарекомендовавших себя стратегий состоит в использовании динамического шардирования и распределения новых данных по шардам случайным образом. Когда шард заполнен, можно выставить флаг, который говорит приложению, что туда не нужно добавлять информацию. Если в будущем вы захотите снова открыть шард для записи, флаг можно будет сбросить.

Предположим, что установлен новый узел MySQL и на нем размещены 100 шардов. В самом начале флаг всех секций равен `1`, то есть они открыты для записи. Как только в секции накопится достаточно данных (например, 10 000 пользователей), ее флаг сбрасывается в `0`. Если через какое-то время узел окажется недогруженным из-за того, что владельцы забросили свои учетные записи, некоторые шарды можно заново открыть для добавления новых пользователей.

Если в ходе развития приложения были добавлены новые возможности, повысившие нагрузку на каждый шард, или вы просто просчитались при оценке нагрузки, то некоторые шарды можно перенести на новые узлы. Неприятность заключается в том, что на время выполнения этой операции весь шард должен быть переведен в режим «только для чтения» или вообще выведен из оперативного доступа. Вам и вашим пользователям решать, насколько такое решение приемлемо.

Мы используем и другую тактику — создание двух реплик шарда, каждая из которых содержит полную копию его данных. Затем назначаем каждую реплику ответственной за половину данных и прекращаем отправку запросов в исходный шард. В каждой реплике содержатся некоторые данные, которые она не использует. Мы запускаем фоновое задание, удаляющее нежелательные данные, для чего задействуем специальный инструмент, такой как pt-archiver пакета Percona Toolkit. Это просто и уменьшает время простоя фактически до нуля.

Генерация глобально уникальных идентификаторов

По ходу преобразования системы в шардированное хранилище данных зачастую необходимо генерировать глобально уникальные идентификаторы на разных машинах. В монолитной базе для этой цели обычно используют автоинкрементные столбцы, но по умолчанию модификатор `AUTO_INCREMENT` предназначен для использования на одном сервере, где гарантировать уникальность несложно.

Есть несколько способов решения этой проблемы.

- *Использовать конфигурационные параметры auto_increment_increment и auto_increment_offset.* Они говорят серверу MySQL, на какую величину увеличивать автоинкрементный столбец и с какого значения начинать нумерацию. Например,

в простейшем случае, когда есть всего два сервера, можно сконфигурировать их так, что первый начнет нумерацию с `1`, а второй — с `2` (впрочем, значение `0` тоже годится), а увеличивать счетчик каждый сервер будет на 2. Тогда на одном сервере в этом столбце будут только четные числа, а на другом — только нечетные. Эти параметры применяются ко всем таблицам на данном сервере.

Из-за своей простоты и независимости от какого-либо центрального узла этот способ генерации значений стал очень популярен, но он требует внимания при конфигурировании серверов. Очень легко случайно настроить их так, что будут генерироваться дубликаты, особенно если роль сервера меняется в результате добавления новых серверов или в ходе восстановления после сбоя.

- *Создать таблицу на глобальном узле.* Можно создать таблицу с автоинкрементным столбцом в глобальной базе данных и обращаться к ней из приложений для генерации уникальных чисел.
- *Использовать memcached.* В API сервера memcached имеется функция `incr()`, позволяющая в ходе неделимой операции инкрементировать число и вернуть результат. Также можно использовать Redis.
- *Выделять значения сериями.* Приложение может запросить у глобального узла серию чисел, использовать ее, а затем запросить новую.
- *Использовать комбинацию значений.* Чтобы добиться уникальности идентификаторов на каждом сервере, можно использовать комбинацию значений, например номер шарда и увеличивающееся число. Эта методика обсуждалась в предыдущем разделе.
- *Использовать значения GUID.* Вы можете генерировать глобально уникальные значения с помощью функции `UUID()`. Однако обратите внимание: эта функция не реплицируется корректно с помощью покомандной репликации, хотя отлично работает, если ваше приложение выбирает значение в собственную память и затем использует его как литерал в операторах. Значения GUID большие и непоследовательные, поэтому они не станут хорошими первичными ключами для таблиц InnoDB. Подробнее об этом написано в подразделе «Вставка строк в порядке первичного ключа в InnoDB» на с. 212. Кроме того, в MySQL 5.1 и более поздних версиях есть функция `UUID_SHORT()`, обладающая несколькими хорошими свойствами: она возвращает последовательные короткие значения — 64 бит вместо 128.

Если для создания значений вы пользуетесь каким-то глобальным генератором, следите за тем, чтобы эта единственная точка конкурентного доступа не стала узким местом приложения.

Хотя решение на основе memcached может работать очень быстро (десятки тысяч значений в секунду), сервер не сохраняет свое состояние. После каждой перезагрузки сервера memcached придется инициализировать начальное значение. Для этого предстоит найти значение, чаще всего используемое во всех шардах, что может потребовать очень много времени. К тому же затруднительно реализовать эту операцию как неделимую.

Инструменты для выполнения шардирования

Одна из главных задач, стоящих перед вами при проектировании шардированного приложения, — написание кода для опроса нескольких источников данных.

Раскрытие приложению нескольких источников данных без какого бы то ни было абстрагирования считается порочной практикой, поскольку сильно увеличивает сложность кода. Лучше скрыть источники за каким-нибудь уровнем абстракции. Этот уровень мог бы:

- соединяться с нужным шардом и отправлять ему запрос;
- выполнять распределенную проверку согласованности;
- объединять результаты, полученные от нескольких шардов;
- отсылать запросы в разные шарды;
- управлять блокировками и транзакциями;
- создавать шарды (или по крайней мере автоматически обнаруживать новые шарды на лету) и повторно выполнять их балансирование (если есть время для реализации такой функциональности).

Совсем не обязательно создавать всю инфраструктуру шардирования с нуля. Существует несколько инструментов и систем, которые либо уже частично предоставляют необходимую функциональность, либо специально предназначены для реализации шардированной архитектуры.

Один слой абстракции базы данных с поддержкой шардов — это Hibernate Shards (http://shards.hibernate.org), расширение Google для основанной на Java библиотеки Hibernate для объектно-реляционного сопоставления (ORM) с открытым исходным кодом. Оно обеспечивает реализацию интерфейсов Hibernate Core, поэтому для использования шардированного хранилища приложение не обязательно переписывать. На самом деле ему даже не нужно знать, что оно работает с таким хранилищем. Для распределения данных по секциям применяется стратегия фиксированного распределения.

Еще одна система шардирования называется HiveDB (http://www.hivedb.org).

В PHP вы можете использовать систему Shard-Query написанную Джастином Сванхартом (Justin Swanhart) (http://code.google.com/p/shard-query/), которая автоматически разделяет запросы на составляющие, выполняет их параллельно и объединяет результаты. Коммерческие системы, ориентированные на аналогичные варианты использования, — ScaleBase (http://www.scalebase.com), ScalArc (http://www.scalarc.com) и dbShards (http://www.dbshards.com).

Система Sphinx предназначена для полнотекстового поиска, а не для хранения и извлечения секционированных данных, тем не менее для некоторых видов запросов к шардированному хранилищу она полезна. Она умеет параллельно опрашивать удаленные системы и объединять результаты (дополнительную информацию о Sphinx см. в приложении Е).

Масштабирование с помощью консолидации

Архитектура, использующая шарды, позволяет увеличить отдачу от вашего оборудования. Наши исследования и опыт показывают, что MySQL не может на 100 % использовать возможности современного оборудования. При масштабировании системы до более чем 24 ядер процессора эффективность MySQL перестает расти. Аналогичная ситуация складывается за пределами 128 Гбайт памяти, и MySQL не может даже приблизиться к использованию максимума возможностей ввода/вывода высокопроизводительных флеш-устройств PCIe, таких как карты Virident и Fusion-io. Кроме реализации единственного экземпляра сервера на мощной машине, есть и другой вариант. Вы можете сделать свои шарды довольно маленькими и поместить по нескольку на машину (практика, которая нам кажется разумной в любом случае). При этом следует запускать несколько экземпляров на каждом сервере, деля физические ресурсы сервера таким образом, чтобы каждому экземпляру досталась какая-то часть.

Этот прием действительно работает, хотя мы хотели бы обойтись без него. Это сочетание методов вертикального и горизонтального масштабирования. Того же результата можно добиться и другими способами — вам не нужно использовать шардирование, но оно естественным образом подходит для консолидации на больших серверах.

Некоторые любят добиваться консолидации с помощью виртуализации, в чем есть свои преимущества. Но во многих случаях виртуализация также требует довольно больших затрат производительности. Затраты зависят от технологии, но обычно они весьма велики, и накладные затраты особенно возрастают, когда операции ввода/вывода выполняются очень быстро. В качестве альтернативы можно запустить несколько экземпляров MySQL, каждый из которых прослушивает разные сетевые порты или привязывается к различным IP-адресам.

Нам удалось достичь коэффициента консолидации 10x или 15x на мощном оборудовании. Чтобы найти подходящий вариант, вам придется сравнивать затраты на сложность администрирования с выгодой от повышения производительности.

На данный момент, скорее всего, узким местом станет сеть — это проблема, с которой большинство пользователей MySQL сталкиваются нечасто. Вы можете решить ее, используя несколько связанных сетевых адаптеров. Подходит ли для этого ядро Linux, зависит от версии, поскольку более старые ядра могут использовать только один процессор для сетевых прерываний на каждое связанное устройство. Так что не следует соединять слишком много кабелей с немногочисленными виртуальными устройствами, иначе вы столкнетесь с другим узким местом сети внутри ядра. Новые ядра должны помочь в этом, поэтому проверьте разводку кабеля, чтобы узнать, какими возможностями вы располагаете.

Еще один способ добиться значительного эффекта от этой стратегии — связать каждый экземпляр MySQL с конкретными ядрами. Это объясняется двумя причинами: во-первых, тем, что MySQL обеспечивает более высокую производительность в пересчете на одно ядро при меньшем количестве ядер, поскольку внутренний

потенциал масштабируемости ядра ограничен, а во-вторых, тем, что, когда экземпляр работает с потоками на нескольких ядрах, синхронизация общих данных между ядрами уменьшает накладные затраты. Это помогает избежать ограничений масштабируемости самого оборудования. Подключение MySQL лишь к нескольким ядрам может уменьшить перекрестные запросы между ядрами ЦП. Обратили внимание на повторяющуюся тему? Подключите процесс к ядрам, которые находятся на одном физическом сокете, чтобы получить наилучшие результаты.

Масштабирование кластеризацией

Сценарий мечты для масштабирования — это единая логическая база данных, которая может хранить столько данных, сколько вам нужно, обслуживать столько запросов, сколько требуется, и увеличиваться настолько, насколько вы хотите. Многие люди в первую очередь думают о создании кластера или сетки, которые легко справляются с этим, из-за чего приложению не нужно делать никакой грязной работы. В то же время оно будет знать, что данные располагаются на многих серверах, а не на одном. С появлением облака становится интересным также автомасштабирование — динамическое добавление серверов или удаление их из кластера в ответ на изменения рабочей нагрузки или размера данных. Во втором издании этой книги мы выразили сожаление в связи с тем, что доступная технология в действительности не соответствует этой задаче. С тех пор было много шума по поводу так называемых NoSQL-технологий. Многие сторонники NoSQL делали странные и необоснованные утверждения, например: «Реляционная модель не может масштабироваться» или «SQL не может масштабироваться». Появились новые концепции, и все стали провозглашать новые лозунги. Кто в наши дни не слышал о согласованности в конечном счете, BASE, векторных часах или теореме CAP?

По прошествии времени справедливость была хотя бы частично восстановлена. Опыт показывает, что многие базы данных NoSQL по-своему примитивны и сами не могут решать множество задач[1]. В то же время возникло множество технологий на базе SQL — то, что Мэтт Аслетт (Matt Aslett) из группы 451 называет базами данных NewSQL. В чем же разница между SQL и NewSQL? Базы данных NewSQL устанавливают, чтобы доказать, что SQL и реляционная технология не являются проблемой. Скорее, проблемы с масштабируемостью в реляционных базах данных являются проблемами реализации, а новые реализации показывают лучшие результаты.

Все новое — это хорошо забытое старое? И да и нет. Многие высокопроизводительные реализации кластерных реляционных баз данных построены на низкоуровневых строительных блоках, которые выглядят похожими на базы данных NoSQL, особенно хранилища «ключ — значение». Например, NDB Cluster не является базой данных SQL — это масштабируемая база данных, к которой можно получить доступ через свой собственный API, в котором очень много от NoSQL, но который также может

[1] Да-да, мы знаем, выбирайте правильный инструмент для работы. Можете вставить сюда еще одну очевидную, но подходящую цитату.

понимать команды SQL, когда вы вместе с ним используете подсистему хранения MySQL. Это полностью распределенный, без разделения ресурсов, высокопроизводительный, автошардированный, транзакционный сервер базы данных без единой точки отказа. Это очень продвинутая база данных, которая не имеет равных для достижения конкретных целей. И за последние несколько лет она стала намного более мощной, сложной и универсальной. В то же время базы данных NoSQL постепенно начинают все больше походить на реляционные базы данных, а некоторые даже разрабатывают языки запросов, которые напоминают SQL. Типичная кластерная база данных будущего, вероятно, будет немного похожа на смесь SQL и NoSQL, с несколькими механизмами доступа для разных вариантов использования. Итак, мы учимся у NoSQL, но и SQL остается в разряде кластеризованных баз данных.

На момент написания книги инструменты технологий кластерной или распределенной базы данных, близкие к MySQL, — это, например, NDB Cluster, Clustrix, Percona XtraDB Cluster, Galera, Schooner Active Cluster, Continuent Tungsten, ScaleBase, ScaleArc, dbShards, Xeround, Akiban, VoltDB и GenieDB. Все они более или менее встроены, доступны через MySQL или связаны с ней. Рассмотрим некоторые из них далее в этой книге, например, поговорим о Xeround в главе 13. Еще в главе 10 мы обсудили Continuent Tungsten и несколько других технологий, но и здесь отведем немного места паре из них.

Прежде чем начать, нужно еще раз подчеркнуть, что масштабируемость, высокая доступность, транзакционная способность и т. д. являются независимыми друг от друга свойствами систем баз данных. Некоторые ошибочно рассматривают их как одно и то же, но это не так. В этой главе мы фокусируемся на масштабируемости. Однако в реальной жизни масштабируемая база данных не очень хороша, если она не высокопроизводительна. Кроме того, вряд ли кто-то хочет масштабироваться без высокой доступности и пр. Определенная комбинация всех этих приятных свойств — это святой Грааль баз данных, который очень трудно найти. Однако это выходит за рамки данной главы.

Наконец, большинство кластерных продуктов NewSQL появились сравнительно недавно (за исключением NDB Cluster). Мы встречались с ними не настолько часто, чтобы хорошо знать их сильные и слабые стороны. И хотя они могут использовать протокол связи MySQL или иначе быть связанными с MySQL, все же это не MySQL, поэтому действительно выходят за рамки этой книги. Так что просто упомянем их и предоставим вам делать выводы об их пригодности для использования.

MySQL Cluster (NDB Cluster)

MySQL Cluster представляет собой комбинацию двух технологий: базы данных NDB и подсистемы хранения MySQL в качестве интерфейса SQL. NDB — это распределенная отказоустойчивая база данных без разделения ресурсов, которая предлагает синхронную репликацию и автоматическое секционирование данных по узлам. Подсистема хранения NDB Cluster преобразует SQL в вызовы API NDB и выполняет операции внутри сервера MySQL, когда их нельзя передать для выполнения NDB.

(NDB является хранилищем данных с парой «ключ — значение» и не может выполнять сложные операции, такие как соединения и агрегация.)

NDB — это сложная база данных, имеющая очень мало общего с MySQL. Для использования NDB MySQL не нужна: NDB можно запускать как автономный сервер базы данных для пар «ключ — значение». Ее сильные стороны включают чрезвычайно высокую пропускную способность для записи и поиска по ключу. NDB автоматически определяет, какой узел должен содержать заданное значение, основанное на хеше ключа. Когда вы получаете доступ к NDB через MySQL, первичный ключ строки является ключом, а остальная часть строки — значением.

Поскольку NDB основана на совершенно новом наборе технологий и поскольку кластер отказоустойчив и распределен, администрировать ее правильно совсем не просто. Есть много динамических элементов, и, чтобы избежать проблем, такие операции, как обновление кластера или добавление узла, нужно выполнять определенным образом. NDB — технология с открытым исходным кодом, но вы можете приобрести коммерческую поддержку от Oracle. Часть этой подписки является запатентованным продуктом Cluster Manager, который помогает автоматизировать многие утомительные и сложные задачи. (Severalnines также предлагает продукт управления кластерами, см. http://www.severalnines.com.)

MySQL Cluster быстро приобретает все больше возможностей и способностей. Например, недавно выпущенные версии поддерживают больше типов изменений в кластере без простоя и могут выполнять некоторые виды запросов на узлах, где хранятся данные. Тем самым уменьшается необходимость в передаче данных по сети и выполнении запросов внутри MySQL. (Эта функция была переименована из `push-down joins` в `adaptive query localization`.)

Ранее в NDB был реализован профиль производительности, который абсолютно отличался от других подсистем хранения MySQL, однако последние версии стали более универсальны. Таким образом, она становится хорошим решением для все большего числа приложений, в том числе игр и мобильных приложений. Мы должны подчеркнуть, что NDB — это серьезная технология, которая позволяет использовать одно из крупнейших приложений, предназначенное для решения важных задач при чрезвычайно высоких нагрузках, требующее задержки и безотказной работы. Например, практически любой телефонный звонок, совершенный в сотовой сети в любой точке мира, использует NDB, и не просто случайным образом — это важнейшая база данных для многих сотовых провайдеров.

NDB нужна быстрая и надежная сеть для подключения узлов, а для достижения лучшей производительности желательно иметь специальные высокоскоростные внутренние соединения. Кроме того, она работает в основном в памяти, поэтому для нее требуется большой объем памяти на серверах.

Каковы же ее недостатки? Пока еще она не очень хорошо работает при сложных запросах, в которых используется много соединений или агрегаций. Кроме того, не рассчитывайте на ее применение для хранения данных. Это транзакционная система, но она не поддерживает MVCC, и при чтении возникают блокировки. Она также не об-

наруживает взаимных блокировок. При возникновении взаимной блокировки NDB пытается решить проблему посредством ожидания. Есть еще множество тонкостей и моментов, о которых вы должны знать. Однако все они заслуживают отдельной книги. (Книги по MySQL Cluster есть, однако большинство из них устарело. Лучше всего обратиться к руководству пользователя.)

Clustrix

Clustrix (http://www.clustrix.com) — это распределенная база данных, которая понимает протокол MySQL, так что ее можно считать упрощенной заменой MySQL. Однако если не брать в расчет протокол, то это совершенно новая технология, построенная не на основе MySQL. Это полностью ACID, транзакционная база данных SQL с поддержкой MVCC, ориентированная на рабочие нагрузки OLTP. Clustrix секционирует данные по узлам для отказоустойчивости и распределяет запросы к данным, которые выполняются параллельно на узлах, вместо того чтобы извлекать данные из узлов хранения в централизованный узел выполнения. Кластер расширяется онлайн с помощью добавления узлов для обработки большего количества данных или большей нагрузки. В некотором роде Clustrix похожа на MySQL Cluster. Ключевыми отличиями являются полностью распределенное выполнение и отсутствие верхнего уровня прокси или координатора запросов перед кластером. Clustrix естественным образом понимает протокол MySQL, поэтому MySQL не нужно переводить команды из своего протокола в протокол Clustrix. Напротив, кластер MySQL на самом деле объединяет три компонента: MySQL, подсистему хранения NDBCLUSTER и NDB.

Выполненные нами лабораторные исследования и эталонное тестирование подтверждают, что Clustrix предлагает высокую производительность и масштабируемость. Clustrix выглядит очень перспективной технологией, так что мы пристально следим за ней.

ScaleBase

ScaleBase (http://www.scalebase.com) — это программа прокси, которая находится между вашим приложением и несколькими внутренними серверами MySQL. Она разбивает входящие запросы на части, распределяет их для одновременного выполнения на внутренних серверах и собирает результаты для доставки обратно в приложение. Однако до сих пор мы с ней не работали. Конкурирующие технологии — ScaleArc (http: //www.scalearc.com) и dbShards (http://www.dbshards.com).

GenieDB

GenieDB (http://www.geniedb.com) родилась как хранилище документов NoSQL для географически распределенного развертывания. Теперь в ней реализован уровень SQL, доступный через подсистему хранения MySQL. Она построена на наборе технологий, включая локальный кэш в памяти, уровень обмена сообщениями и постоянное дисковое хранилище данных. Они работают вместе, чтобы обеспечить для приложения возможность быстрого выполнения запросов к локальным данным с помощью

гарантий нестрогой согласованности в конечном счете или к распределенному кластеру (с добавленной задержкой сети), и таким образом гарантировать просмотр наиболее новых данных.

Уровень совместимости с MySQL с помощью подсистемы хранения не обеспечивает 100 % возможностей MySQL, но может поддерживать такие готовые приложения, как Joomla!, Word-Press и Drupal. Вариант использования для подсистемы хранения MySQL таков: сделать GenieDB доступной наряду с подсистемой хранения данных ACID, такой как InnoDB. GenieDB не является базой данных ACID.

Мы сами не работали с GenieDB и не видели каких-либо производственных развертываний.

Akiban

Вероятно, Akiban (http://www.akiban.com) лучше всего можно описать как ускоритель запросов. Она физически хранит данные, чтобы соответствовать паттернам запросов, что позволяет выполнять соединения между таблицами с гораздо меньшими затратами. Хотя это и похоже на денормализацию, расположение данных не является избыточным, поскольку это не то же самое, что предварительное вычисление соединений и сохранение результатов. Вместо этого кортежи из соединенных таблиц чередуются друг с другом, поэтому их можно просматривать последовательно в порядке соединения. Для этого требуется, чтобы администратор определил паттерны запросов, которые выиграли бы от так называемой методики группировки таблиц, и спроектировал оптимальные группы таблиц для таких запросов. В настоящее время системная архитектура предполагает настройку Akiban таким образом, чтобы она использовалась для репликации данных с главного сервера MySQL и обслуживания запросов, которые в противном случае выполнялись бы медленно. Предполагается, что тем самым будет достигнуто увеличение скорости на пару порядков. Однако мы не видели производственных развертываний и не проводили лабораторных исследований[1].

Обратное масштабирование

Один из самых простых способов справиться с увеличившимся объемом данных и ростом рабочей нагрузки состоит в том, чтобы архивировать и удалять ставшую ненужной информацию. В зависимости от рабочей нагрузки это может дать весьма ощутимый эффект. Конечно, такой подход не отменяет другие стратегии масштабирования, но в качестве краткосрочной стратегии, позволяющей выиграть время, годится. Да и в более долгосрочной перспективе он может применяться для борьбы с ростом объема данных.

---

[1] Возможно, мы немного жульничаем, включая Akiban в список кластерных баз данных, поскольку она таковой не является. Однако в некоторых моментах она очень похожа на множество других баз данных NewSQL.

При проектировании стратегии архивирования и удаления данных нужно принимать во внимание следующие факторы.

- *Влияние на приложение.* Хорошая стратегия архивирования позволяет убрать данные с сильно нагруженного OLTP-сервера, не оказывая заметного влияния на обработку транзакций. Основная идея состоит в том, что сначала нужно найти подлежащие удалению строки, а затем удалять их небольшими порциями. Обычно необходимо отыскать такое соотношение между количеством строк в одной порции и размером транзакции, которое обеспечивает приемлемый баланс между уровнем конкурентности и транзакционными издержками. Задания архивирования должны быть написаны так, чтобы при необходимости они уступали сервер транзакционным задачам.
- *Какие строки архивировать.* Если вы точно знаете, что некоторые данные больше не понадобятся, можете удалить или архивировать их. Однако можно спроектировать приложение и так, чтобы оно архивировало данные, к которым редко обращаются. Архивированные данные можно хранить в отдельных таблицах и обращаться к ним через представления или даже полностью переместить на другой сервер.
- *Поддержание согласованности данных.* Наличие связей между данными может усложнить архивирование и удаление. Хорошо спроектированная программа архивирования сохраняет логическую согласованность информации, по крайней мере в той степени, в которой это нужно приложению, не открывая гигантские транзакции с несколькими таблицами.

  При наличии связей принять решение о том, какие таблицы архивировать в первую очередь, всегда трудно. Необходимо учитывать возможность появления в ходе архивирования «осиротевших» или «овдовевших» строк. Обычно вы должны либо пойти на нарушение ограничений внешних ключей (в InnoDB отключить такие ограничения можно командой `SET FOREIGN_KEY_CHECKS=0`), либо оставить на время записи с «висячими» указателями. Что именно предпочесть, зависит от того, как приложение обращается к данным. Если программа просматривает набор взаимосвязанных таблиц сверху вниз, то и архивировать их следует в таком же порядке. Например, если заказы всегда читаются до счетов-фактур, то сначала архивируйте заказы: приложение не должно видеть «осиротевших» счетов-фактур, а архивировать их можно будет позже.
- *Как избежать потери данных.* Если архивируются данные, расположенные на нескольких серверах, то не стоит применять распределенные транзакции, тем более что архивирование может производиться в таблицы MyISAM или иного нетранзакционного типа. Поэтому, чтобы избежать потери данных, следует сначала вставить значения в конечную таблицу и лишь потом удалять их из исходной. Кроме того, разумно попутно записывать архивированные данные в файл. Программа архивирования должна быть спроектирована так, чтобы ее можно было в любой момент прервать, а затем перезапустить и это не приводило бы к рассогласованности данных или повреждению индексов.

- *Разархивация.* Зачастую можно убрать значительно больше данных, если процедура архивации дополняется стратегией разархивации. Такой подход позволяет архивировать данные, не будучи уверенными в том, что они больше не понадобятся, поскольку всегда есть возможность в случае необходимости восстановить их. Если удается выявить несколько точек, в которых система может проверить, нужно ли извлечь архивированные данные, то такую стратегию реализовать довольно просто. Например, если архивируются неактивные пользователи, такой точкой может быть процедура входа в систему. Если вход невозможен из-за отсутствия пользователя, то программа посмотрит, нет ли такого пользователя в архиве, и, если есть, извлечет его оттуда и продолжит выполнение процедуры входа.

В пакете Percona Toolkit имеется инструмент pt-archiver, помогающий эффективно архивировать и очищать таблицы MySQL. Однако разархивирования он не поддерживает.

Отделение активных данных. Даже если вы не перемещаете устаревшую информацию на другой сервер, производительность многих приложений может увеличиться, если отделить активные данные от неактивных. Это повышает эффективность использования кэша и позволяет применять для активных и неактивных данных различные аппаратные и программные архитектуры. Далее перечислены несколько способов решения этой задачи.

- *Разбиение таблицы на несколько частей.* Часто имеет смысл разбивать таблицу на части, особенно если она не помещается в память целиком. Например, таблицу `users` можно разделить на `active_users` и `inactive_users`. Возможно, вам кажется, что это не обязательно, поскольку в кэше базы данных в любом случае задерживаются только часто используемые (горячие) данные, однако это зависит от подсистемы хранения. В InnoDB единицей кэширования является страница. Если на одной странице умещаются 100 пользователей, лишь 10 % из которых активны, то с точки зрения InnoDB каждая страница будет горячей, и тем не менее 90 % данных на странице — пустая трата памяти. Разбиение таблицы могло бы существенно улучшить использование ОЗУ.
- *Секционирование на уровне MySQL.* В версии MySQL 5.1 появился механизм секционирования таблиц, который может помочь удерживать недавно использовавшиеся данные в памяти. Подробнее о секционировании см. в главе 7.
- *Секционирование данных по времени.* Если приложение постоянно получает новые данные, то вполне вероятно, что недавние данные будут гораздо активнее более старых. Например, нам известна служба блогов, в которой основной трафик порождают сообщения и комментарии, созданные за последние семь дней. Большая часть обновлений также относится к этим данным. В результате разработчики хранят информацию за последнюю неделю целиком в памяти и применяют репликацию для создания копии на другом диске на случай сбоя. Прочая информация постоянно хранится в другом месте.

Нам также встречались системы, в которых данные, относящиеся ко всем пользователям, содержатся в шардах на двух узлах. Новые значения поступают на активный узел, на котором очень много памяти и быстрые диски. Эти данные оптимизированы для очень быстрого доступа. Второй узел, где хранятся более старые данные, оснащен очень большими, но сравнительно медленными дисками. Приложение исходит из предположения, что обращения к старым записям маловероятны. Для большинства приложений это вполне здравое предположение, позволяющее удовлетворить 90 % и даже более запросов на основе лишь 10 % данных, поступивших в последнее время.

Такую стратегию несложно реализовать посредством динамического шардирования. Например, определение таблицы с каталогом шардов могло бы выглядеть следующим образом:

```
CREATE TABLE users (
    user_id           int unsigned not null,
    shard_new         int unsigned not null,
    shard_archive     int unsigned not null,
    archive_timestamp timestamp,
    PRIMARY KEY (user_id)
);
```

Скрипт архивирования может перемещать старые данные с активного узла на архивный, обновляя по ходу дела столбец `archive_timestamp`. В столбцах `shard_new` и `shard_archive` хранится информация о том, в шардах с какими номерами находятся новые и архивированные данные соответственно.

Балансирование нагрузки

Основная идея балансирования нагрузки проста: максимально равномерно распределить ее по нескольким серверам. Обычно это решается путем внедрения балансировщика нагрузки (часто он представляет собой специализированное оборудование) перед серверами. Балансировщик передает запрос на установление соединения наименее загруженному из имеющихся серверов. На рис. 11.9 изображена типичная схема балансирования нагрузки для большого сайта, когда один балансировщик обслуживает HTTP-трафик, а другой — MySQL-трафик.

Балансированием нагрузки обычно направлено на достижение следующих пяти целей.

- *Масштабируемость.* Балансирование нагрузки может помочь при определенных типах стратегий масштабируемости, таких как разделение чтения и записи для чтения с подчиненных серверов.
- *Эффективность.* Балансирование нагрузки позволяет более эффективно использовать ресурсы, поскольку вы можете управлять тем, как маршрутизируются запросы. Это особенно важно, когда используются серверы разной мощности — на более мощные можно направлять больше работы.

Рис. 11.9. Типичная архитектура балансирования нагрузки для сайта с большим количеством запросов на чтение

- *Доступность.* Интеллектуальный балансировщик нагрузки передает запросы только серверам, доступным в данный момент.
- *Прозрачность.* Клиентам не нужно знать о том, как сконфигурировано балансирование нагрузки. Им все равно, сколько машин находится за балансировщиком и как они называются, — балансировщик представляет внешним клиентам единый виртуальный сервер.
- *Согласованность.* Если приложение наделено состоянием (транзакции базы данных, сеансы сайта и т. д.), то сам балансировщик должен направлять взаимосвязанные запросы одному и тому же серверу, чтобы состояние не утрачивалось. Это освобождает приложение от обязанности следить за тем, с каким сервером оно соединилось.

В мире MySQL балансирование нагрузки часто применяется в сочетании с шардированием и репликацией. Можно как угодно комбинировать эти технологии и размещать их на том уровне, где это наиболее удобно приложению. Например, можно балансировать нагрузку между несколькими узлами в кластере MySQL. Или балансировать нагрузку между центрами обработки данных, а внутри каждого из них применять шардирование между узлами, каждый из которых является реплицируемой парой «главный — главный с несколькими подчиненными серверами», нагрузка

между которыми также балансируется. То же относится и к стратегии обеспечения высокой доступности — переключение при отказе может быть организовано на нескольких уровнях архитектуры.

У технологии балансирования нагрузки есть много нюансов. Например, как управлять политиками чтения-записи? Некоторые балансировщики делают это самостоятельно, а другие требуют, чтобы приложение знало, какие узлы доступны только для чтения, а какие — еще и для записи.

Вы должны учитывать это, принимая решение о том, как реализовать балансирование нагрузки. Решений существует множество, начиная от одноранговых реализаций типа Wackamole (http://www.backhand.org/wackamole/) до подходов на основе системы доменных имен (DNS), виртуального сервера Linux (Linux Virtual Server, LVS, http://www.linuxvirtualserver.org), аппаратных балансировщиков, MySQL Proxy и управления балансированием нагрузки из приложения.

Большинство наших клиентов, по-видимому, пользуются методикой, предполагающей применение специального оборудования для балансирования нагрузки. Многие из них задействуют HAProxy (http://haproxy.1wt.eu), который, кажется, очень популярен и работает довольно хорошо. Некоторые используют другой TCP-прокси, например Pen (http://siag.nu/pen/). Мы видим, что MySQL Proxy применяется не слишком часто.

Прямое подключение

Некоторые подсознательно ассоциируют балансирование нагрузки с центральной системой, которая устанавливается между приложением и серверами MySQL. Но это не единственный вариант. Можно одновременно сбалансировать нагрузку и сохранить возможность прямого подключения к серверу MySQL из приложения. На самом деле централизованные системы балансирования обычно хорошо работают только тогда, когда имеется пул серверов, которые с точки зрения приложения эквивалентны. Если приложение должно принимать решение, например, безопасно ли читать данные с подчиненного сервера, то обычно требуется уметь напрямую подключаться к этому серверу.

Принятие решений о балансировании нагрузки в приложении позволяет не только учесть особые случаи, но и добиться очень высокой эффективности. Например, если имеются два идентичных подчиненных сервера, то можно на одном из них выполнять все запросы к некоторому подмножеству шардов, а на другом — запросы ко всем остальным шардам. Это позволит эффективно использовать память подчиненных серверов, так как каждому придется кэшировать лишь часть информации на диске. В случае отказа одного из них второй по-прежнему располагает всеми данными, необходимыми для обслуживания запросов ко всем шардам.

В следующих разделах обсуждаются некоторые общепринятые способы прямого подключения из приложения и факторы, которые нужно принимать во внимание при выборе того или иного варианта.

Распределение запросов на чтение и запись при репликации

Механизм репликации в MySQL позволяет поддерживать несколько копий данных и решать, где выполнять запросы — на главном или подчиненном сервере. Основная сложность состоит в том, что делать с неактуальностью данных на подчиненном сервере, неизбежной в силу асинхронной природы репликации. Кроме того, подчиненные серверы должны использоваться только для чтения, тогда как главный может выполнять запросы любого вида.

Обычно приложение приходится модифицировать с учетом этих соображений. Приложение может посылать все запросы на запись главному серверу, а запросы на чтение распределять между главным и подчиненными серверами. В случае, когда некоторое устаревание данных несущественно, можно читать с подчиненных серверов, а если необходимы гарантированно актуальные данные, то с главного. Мы называем такой подход *распределением чтения-записи*.

Те же рассуждения применимы к паре «главный — главный» с активным и пассивным главными серверами, хотя в данном случае запросы на запись следует отправлять только активному серверу. Чтение можно выполнять и на пассивном сервере, если потенциальное устаревание не страшно.

Самая серьезная проблема — как избежать аномалий при чтении неактуальных данных. Классическая ситуация такого рода возникает, когда пользователь вносит некое изменение, скажем, добавляет комментарий в блог, затем перезагружает страницу и не видит этого изменения, поскольку приложение прочитало неактуальные данные с подчиненного сервера.

Далее перечислены некоторые часто применяемые методы распределения запросов на чтение и запись.

- *Распределение с учетом запроса.* Самый простой способ — направлять активному или главному серверу все запросы на запись, а также те запросы на чтение, для которых неактуальные данные неприемлемы. Остальные запросы на чтение попадают на подчиненный или пассивный сервер. Эту стратегию легко реализовать, но на практике подчиненный сервер будет задействован не в полной мере, поскольку не так уж много существует запросов, которым всегда безразлична актуальность данных.
- *Распределение по степени неактуальности данных.* Это некоторое усовершенствование стратегии распределения с учетом запроса. От приложения требуется сравнительно немного дополнительной работы, чтобы понять, насколько отстает подчиненный сервер, и решить, являются ли данные слишком устаревшими. Так поступают многие приложения, формирующие отчеты: коль скоро все данные за предыдущий день реплицированы на подчиненный сервер, наличие стопроцентной синхронизации с главным сервером не имеет значения.
- *Распределение с учетом сеанса.* Чуть более сложный алгоритм решения вопроса о том, откуда читать данные, — учитывать, изменял ли их конкретный пользователь. Работающему с приложением не обязательно видеть все модификации, внесенные другими пользователями, но свои он наблюдать должен. Чтобы реали-

зовать это на уровне сеанса, достаточно установить на нем флажок, который показывает, что в ходе сеанса было произведено изменение, и в течение некоторого времени после этого события направлять запросы на чтение, исходящие от этого пользователя, главному серверу. Обычно мы рекомендуем данную стратегию нашим клиентам, поскольку она является хорошим компромиссом между простотой и эффективностью.

Этот прием можно сочетать с мониторингом отставания репликации: если пользователь изменил какие-то данные 10 секунд назад и ни один подчиненный сервер не отстает более чем на 5 секунд, то читать с подчиненного сервера безопасно. Очень разумно было бы выбрать какой-то один подчиненный сервер и задействовать его на протяжении всего сеанса, в противном случае пользователь будет наблюдать странные эффекты, вызванные тем, что некоторые подчиненные серверы отстают больше других.

- *Распределение с учетом версии.* Эта стратегия похожа на распределение с учетом сеанса: можно отслеживать номера версий и/или временные метки объектов и читать номер версии или временную метку с подчиненного сервера, чтобы понять, насколько свежие на нем данные. Если информация на подчиненном сервере сильно устарела, то можно обратиться за актуальными данными к главному серверу. Можно также увеличивать номер версии объекта верхнего уровня даже в том случае, когда сам объект не изменяется, — это упрощает проверку на устаревание (необходимо заглядывать только в одно место — в объект верхнего уровня). Например, можно обновлять версию пользователя, если он помещает в свой блог новую запись. В таком случае запросы на чтение будут направляться главному серверу.

  Чтение версии объекта с подчиненного сервера означает дополнительные накладные расходы, которые можно уменьшить за счет кэширования. Вопрос о кэшировании и версиях объектов мы рассмотрим в следующих главах.

- *Глобальное распределение с учетом версии или сеанса.* Это вариация на тему распределений с учетом версии и сеанса. Производя операцию записи, приложение выполняет запрос `SHOW MASTER STATUS` после коммита транзакции. Координаты в журнале главного сервера сохраняются в кэше и считаются модифицированным номером версии объекта и/или сеанса. Затем, когда приложение подключается к подчиненному серверу, оно выполняет команду `SHOW SLAVE STATUS` и сравнивает координаты подчиненного сервера с версией, которую запомнил. Если подчиненный сервер дошел хотя бы до той точки, где главный закоммитил транзакцию, то читать с него безопасно.

В большинстве решений по распределению запросов на чтение и запись необходимо следить за отставанием подчиненного сервера и, исходя из этого, решать, куда отправлять запрос. Но имейте в виду, что столбец `Seconds_behind_master` в результатах команды `SHOW SLAVE STATUS` — ненадежный способ определения отставания подчиненного сервера (см. главу 10). Инструмент pt-heartbeat из пакета Percona Toolkit может помочь как в мониторинге задержки репликации, так и в сохранении метаданных, таких как позиция в двоичном журнале. Тем самым облегчается реализация некоторых рассмотренных стратегий.

Если вам неважно, сколько оборудования потребуется для обслуживания нагрузки, то можно упростить себе жизнь и вовсе не пользоваться репликацией для масштабирования. Это поможет избежать сложностей, связанных с распределением запросов между главным и подчиненными серверами. Некоторые считают такой подход оправданным, другие видят в нем расточительное использование оборудования. Такое различие во мнениях отражает разные цели: чего именно требуется достичь — только масштабируемости или еще и эффективности? Если одной из целей является эффективность и, следовательно, хотелось бы задействовать подчиненные серверы не только для хранения копий данных, то, вероятно, вам придется пойти на увеличение сложности.

Изменение конфигурации приложения

Один из способов распределения нагрузки — изменение конфигурации приложения. Например, можно настроить несколько машин так, чтобы они разделяли между собой бремя генерации больших отчетов. В этом случае разные компьютеры подключаются к разным серверам MySQL и каждый генерирует отчет для каждого *N*-го заказчика или сайта.

Реализовать подобную систему обычно совсем не сложно, но если при этом потребуется вносить изменения в код, в том числе в конфигурационные файлы, то система станет слишком хрупкой и громоздкой. Любое решение, требующее изменений программного кода на каждом сервере или даже одного изменения в каком-то центральном пункте, откуда оно публикуется путем копирования файлов или команд обновления в системе управления версиями, принципиально порочно. В то же время, если конфигурация хранится в базе данных и/или в кэше, можно обойтись без публикации изменений, внесенных в код.

Изменение доменных имен

Создание доменных имен для разных целей — это очень прямолинейная техника балансирования нагрузки, которая тем не менее неплохо работает в некоторых простых приложениях. Затем по мере необходимости вы можете указать имена на разных серверах. В простейшем случае имеется одно доменное имя для серверов, допускающих только чтение, и еще одно — для сервера, на котором выполняются операции записи. Если подчиненные серверы успевают за главным, то можно изменить первое доменное имя так, чтобы оно указывало на подчиненные серверы: когда они начинают отставать, это имя перенаправляется на главный сервер.

Методику на основе доменных имен очень легко реализовать, но и недостатков у нее масса. И самый главный — то, что система DNS вам не полностью подконтрольна.

- Изменения в системе DNS не мгновенны и не неделимы. Требуется довольно длительное время для того, чтобы изменение распространилось по сети или между сетями.
- Данные DNS кэшируются в разных местах, и срок хранения является рекомендательным, а не обязательным.

- Для того чтобы изменение в системе DNS было воспринято полностью, возможно, придется перезагрузить приложение или сервер.
- Не стоит использовать несколько IP-адресов для одного доменного имени и полагаться на алгоритм циклического перебора для балансирования запросов. Поведение этого алгоритма не всегда предсказуемо.
- Администратор базы данных не всегда имеет прямой доступ к системе DNS.

Если ваше приложение не слишком простое, то довольно опасно полагаться на систему, которая не полностью контролируема. Степень контролируемости немного повысится, если вносить изменения в файл `/etc/hosts`, а не в систему DNS. Можно с уверенностью сказать, что изменение вступило в силу сразу после модификации этого файла. Это лучше, чем дожидаться, пока истечет срок хранения DNS-записи в кэше, но все равно неидеально.

Обычно мы не рекомендуем создавать приложения, хоть как-то зависящие от DNS. Лучше избегать такой зависимости даже в простом ПО, поскольку никто не знает, как оно разрастется в будущем.

Переключение IP-адресов

Некоторые способы балансирования нагрузки базируются на переключении виртуальных IP-адресов[1] между серверами — иногда такая техника прекрасно работает. На первый взгляд это похоже на изменение доменных имен, но на самом деле общего между технологиями мало. Серверы прослушивают сетевой трафик не по доменному имени, а по конкретному IP-адресу, поэтому переключение IP-адресов сохраняет статичные доменные имена. Известить об изменении IP-адреса можно неделимой операцией и очень быстро с помощью команд протокола определения адреса (Address Resolution Protocol, ARP).

Наиболее часто мы видели, как для этих целей применяли технологию Pacemaker, которая является преемницей инструмента Heartbeat проекта Linux-HA. Например, у вас может быть один IP-адрес, связанный с ролью «только для чтения», и он позаботится о том, чтобы по мере необходимости переместить IP-адрес между машинами. Другими инструментами для достижения этой цели являются LVS и Wackamole.

Полезный прием состоит в том, чтобы назначить фиксированный IP-адрес каждому физическому серверу. Этот адрес — принадлежность самого сервера, и он никогда не изменяется. Далее каждой логической службе выделяется виртуальный IP-адрес. Эти адреса могут передаваться от одного сервера другому, следовательно, возникает возможность перемещать экземпляры служб и приложений без переконфигурирования приложения. Это очень удобно, даже если для балансирования нагрузки или обеспечения высокой доступности не приходится часто переключать IP-адреса.

[1] Виртуальный IP-адрес не связан ни с каким конкретным компьютером или сетевым интерфейсом; он мигрирует от одного компьютера к другому.

Представляем посредника

До сих пор мы предполагали, что приложение взаимодействует напрямую с серверами MySQL. Однако во многих схемах балансирования нагрузки встречается посредник, задача которого — выступать в роли прокси-сервера сетевого трафика. Посредник принимает весь трафик и направляет его нужному серверу, при этом ответы сервера доставляются компьютеру, с которого поступил исходный запрос. Иногда посредник реализуется аппаратно, а иногда — программно[1]. Такая архитектура изображена на рис. 11.10. Подобные решения обычно работают очень хорошо, но если сам балансировщик нагрузки не резервирован, то он становится единой точкой отказа. Мы видели множество балансировщиков, успешно используемых в различном программном обеспечении, — от продуктов с открытым исходным кодом, таких как HAProxy, до практически любой известной вам коммерческой системы.

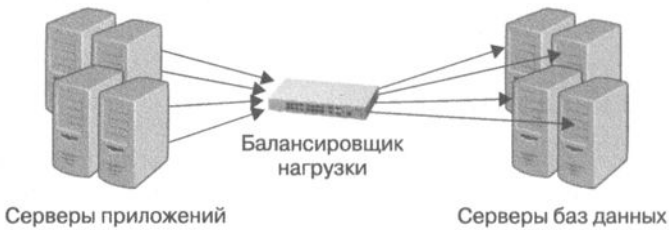


Рис. 11.10. Балансировщик нагрузки, работающий как посредник

Балансировщики нагрузки

На рынке представлено немало аппаратных и программных решений для балансирования нагрузки, но лишь немногие спроектированы специально для серверов MySQL[2]. Гораздо чаще в балансировании нагрузки нуждаются веб-серверы, поэтому во многие универсальные устройства встроена специальная поддержка протокола HTTP и минимальные базовые средства для всего остального. Поскольку соединения с сервером MySQL — это обычные TCP/IP-соединения, то для MySQL можно использовать балансировщики нагрузки общего назначения. Однако отсутствие специальной поддержки MySQL накладывает следующие ограничения.

- Если балансировщик нагрузки не знает об истинной нагрузке на MySQL, то он способен лишь распределять запросы, а не балансировать нагрузку. Не все запросы одинаковы, но универсальный балансировщик обычно не видит различий между ними.

---

[1] Систему LVS можно сконфигурировать так, что она вступает в игру только тогда, когда приложению необходимо создать новое соединение, а после этого не выполняет функции посредника.

[2] MySQL Proxy является исключением, однако ему присущ ряд проблем, таких как узкие места добавленной задержки и масштабируемости.

- Большинство балансировщиков знают о структуре HTTP-запроса и умеют ассоциировать сеанс с сервером, сохраняя тем самым состояние веб-сервера. У соединений MySQL тоже имеется состояние, но балансировщик понятия не имеет, как связать все запросы на соединение, исходящие из одного сеанса, с одним сервером MySQL. Результатом оказывается снижение эффективности (если бы все запросы от одного сеанса попадали одному серверу MySQL, то кэш этого сервера использовался бы более эффективно).
- Наличие пула соединений и постоянных соединений может помешать балансировщику нагрузки распределять запросы на соединение. Пусть, например, пул открывает заданное количество соединений, а балансировщик нагрузки распределяет их между четырьмя серверами MySQL. А теперь предположим, что добавлены еще два сервера. Поскольку пул не запрашивает создание новых соединений, то эти серверы простаивают. Кроме того, соединения из пула могут распределяться между серверами несправедливо, так что некоторые окажутся перегруженными, а другие — недогруженными. Эти проблемы можно частично решить, задавая на различных уровнях срок хранения соединений в пуле, но это сложно и труднореализуемо. Пул соединений работает оптимально, когда самостоятельно занимается балансированием нагрузки.
- Большинство универсальных балансировщиков нагрузки умеют контролировать состояние соединения только для HTTP-серверов. Простой балансировщик может проверить, что сервер принимает соединение с некоторым TCP-портом, — это абсолютно необходимый минимум. Более развитый балансировщик может отправить HTTP-запрос и проанализировать код ответа, чтобы удостовериться в нормальной работе веб-сервера. Но MySQL не принимает HTTP-запросы на порт 3306, так что контролировать работоспособность придется самостоятельно. Можно установить на компьютер, где работает сервер MySQL, еще и HTTP-сервер и направить балансировщик нагрузки на специальный скрипт, который уже проверит состояния MySQL и вернет соответствующий код[1]. Обращать внимание нужно прежде всего на загрузку операционной системы (обычно эти данные доступны в файле `/proc/loadavg`), состояние репликации и количество соединений с сервером MySQL.

Алгоритмы балансирования нагрузки

Существует много разных алгоритмов для определения того, какой сервер должен получить следующий запрос на соединение. Каждый поставщик применяет собственную терминологию в этой области, но из приведенного далее перечня должно быть понятно, какие имеются варианты.

- *Случайное балансирование.* Балансировщик передает соединение серверу, случайно выбранному из пула доступных серверов.
- *Циклическое балансирование.* Балансировщик передает соединения серверам по кругу: A, B, C, A, B, C и т. д.

[1] Если вы способны самостоятельно написать программу, которая будет прослушивать порт 80, и можете сконфигурировать демон xinetd так, чтобы он вызывал вашу программу, то и веб-сервер устанавливать не придется.

- *Минимум соединений.* Следующее соединение передается серверу, с которым в данный момент установлено наименьшее количество соединений.
- *Самый быстрый ответ.* Следующее соединение передается серверу, который быстрее всего обрабатывает запросы. Этот алгоритм неплохо работает, когда в пул включены как быстрые, так и медленные компьютеры. Однако применить его к SQL проблематично, поскольку сложность запросов варьируется в очень широких пределах. Даже один и тот же запрос может вести себя очень по-разному в зависимости от обстоятельств, например от того, обслуживается он с диска или из кэша запросов.
- *Хешированное балансирование.* Балансировщик хеширует IP-адрес источника соединения, отображая его на один из серверов в пуле. Таким образом, все соединения от одного источника поступают одному и тому же серверу. Отображение модифицируется лишь при изменении количества компьютеров в пуле.
- *Взвешенное балансирование.* Балансировщик может применять сразу несколько алгоритмов, приписывая каждому определенный вес. Например, в наличии могут быть компьютеры с одним и двумя процессорами. Машины с двумя процессорами примерно в два раза мощнее тех, что с одним, поэтому балансировщик нагрузки можно сконфигурировать так, чтобы он посылал им в среднем в два раза больше соединений.

Какой алгоритм окажется для вас наилучшим, зависит от рабочей нагрузки. Например, балансирование по минимуму соединений может привести к перегрузке запросами вновь добавленных в пул серверов как раз в тот момент, когда их кэш еще не «прогрет». Авторы первого издания настоящей книги сталкивались с этой проблемой на практике.

Для подбора оптимального для имеющейся рабочей нагрузки решения необходимо экспериментировать. Обязательно протестируйте его не только в нормальных, но и в экстраординарных условиях. Только при необычных обстоятельствах — высокой нагрузке, изменении схемы или ненормально большом количестве серверов, выведенных из эксплуатации, — можно допустить аномальное поведение.

Мы описали только алгоритмы немедленного обслуживания, когда запросы на соединение не ставятся в очередь. Иногда алгоритмы с очередями оказываются более эффективными. Например, можно поддерживать на сервере баз данных заданный уровень конкуренции — скажем, не более N активных транзакций в каждый момент времени. Если транзакций слишком много, то очередной запрос ставится в очередь и обслуживается, когда какой-нибудь сервер станет доступен в соответствии с выбранным критерием. Некоторые пулы соединений поддерживают подобные алгоритмы с очередями.

Добавление серверов в пул и удаление из пула

Чтобы добавить новый сервер в пул, обычно недостаточно просто включить его в сеть и уведомить балансировщик о его существовании. Если вам кажется, что все будет хорошо, если сервер не перегружен запросами на соединение, то это не всегда верно. Иногда можно увеличивать нагрузку на сервер постепенно, но некоторые серверы

с холодным кэшем могут отвечать настолько медленно, что в течение некоторого времени не должны получать вообще никаких запросов. Если на то, чтобы показать страницу пользователю, уходит 30 секунд, то такой сервер нельзя счесть пригодным для использования даже при небольшом объеме трафика. Чтобы решить эту проблему, можно в течение некоторого времени повторять на новом сервере все запросы `SELECT` с какого-нибудь активного сервера и только потом известить о нем балансировщик нагрузки. Для этого достаточно читать и воспроизводить на новом сервере файлы журналов активного сервера либо зафиксировать сетевой трафик работающего сервера и воспроизвести часть его запросов на новом. Инструмент pt-query-digest пакета Percona Toolkit может помочь в этом. Другой хорошей методикой является использование возможностей быстрого «прогрева» в Percona Server или MySQL 5.6.

Конфигурировать серверы, входящие в пул соединений, следует так, чтобы резервной пропускной способности оказалось достаточно на время вывода некоторых серверов для обслуживания и для случая, если какой-то из них откажет. Мощность всех узлов должна быть заложена с запасом.

Убедитесь, что конфигурационные лимиты достаточно высоки для продолжения работы после удаления некоторых серверов из пула. Например, если каждый сервер MySQL обычно обрабатывает 100 соединений, то для параметра `max_connections` должно быть установлено значение `200`. Тогда даже если половина узлов выйдет из строя, тех, что остались в пуле, хватит для обработки такого же количества соединений, что и раньше.

Балансирование нагрузки при наличии одного главного и нескольких подчиненных серверов

Самая распространенная топология репликации — это один главный и несколько подчиненных серверов. Отойти от такой архитектуры не всегда легко. Во многих приложениях предполагается, что существует единственное место для всех операций записи или имеется один сервер, на котором в любой момент присутствуют все данные. Хотя с точки зрения масштабирования такая архитектура не является наилучшей, ее можно с успехом применить, если воспользоваться балансированием нагрузки. В данном разделе рассматриваются некоторые подходы к этому.

- *Функциональное секционирование.* Мощность можно немного увеличить, если сконфигурировать некоторые подчиненные серверы или группы таких серверов для достижения конкретных целей. Например, выделить серверы для генерации отчетов и анализа данных, организации хранилищ и полнотекстового поиска.
- *Фильтрация и секционирование данных.* Данные можно распределить по подчиненным серверам с помощью фильтров репликации (см. главу 10). Эта стратегия хорошо работает, если информация на главном сервере уже находится в разных базах или таблицах. К сожалению, не существует встроенного способа выполнять фильтрацию на уровне строк. Вам нужно будет сделать что-то творческое (читай: хакерское), чтобы реализовать это. Возможно, придется использовать триггеры и множество разных таблиц.

Даже если вы не секционируете данные между подчиненными серверами, эффективность кэша можно повысить за счет секционирования операций чтения вместо того, чтобы распределять их случайно. Например, можно направлять запросы на чтение для пользователей, чьи фамилии начинаются с букв A — M, на один подчиненный сервер, а запросы для остальных пользователей — на другой. Тогда кэш каждого сервера будет использоваться эффективнее, так как более вероятно, что данные для повторяющихся запросов уже находятся в кэше. В лучшем случае, когда операций записи нет вообще, такая стратегия дает кэш, размер которого равен сумме размеров кэшей обоих серверов. Для сравнения: если распределять запросы на чтение между серверами случайным образом, то данные в кэше каждого сервера дублируются, поэтому его полный эффективный размер не превышает размер самого большого кэша на всех подчиненных серверах независимо от их количества.

- *Перенос части операций записи на подчиненный сервер.* Не всегда главный сервер обязан выполнять все операции записи. И главный, и подчиненные серверы можно освободить от значительного объема работы, если разложить запросы на части и некоторые из них выполнять непосредственно на подчиненных серверах. Дополнительную информацию по этой теме см. в главе 10.
- *Гарантированная актуальность данных на подчиненном сервере.* Если на подчиненном сервере требуется запустить некий процесс, которому важно, чтобы данные были актуальными хотя бы на известный момент времени, даже если придется подождать, то можно воспользоваться функцией `MASTER_POS_WAIT()`, которая блокирует процесс до тех пор, пока подчиненный сервер не достигнет желаемой степени синхронизации с главным. Или можно применить для контроля актуальности записи пульсацию репликации (подробнее см. в главу 10).
- *Синхронизация записи.* Можно воспользоваться функцией `MASTER_POS_WAIT()`, которая позволяет гарантировать, что операция записи действительно дошла до одного или нескольких подчиненных серверов. Если приложение желает эмулировать синхронную репликацию для обеспечения гарантированной безопасности данных, то можно перебирать все подчиненные серверы в цикле, выполняя `MASTER_POS_WAIT()` для каждого. Тем самым создается барьер синхронизации, для пересечения которого может потребоваться длительное время, если какие-то из подчиненных серверов сильно отстают; поэтому применять такую технику следует только в случае крайней необходимости. (Если нужно лишь гарантировать, что какой-то сервер получил событие, то можно дожидаться синхронизации именно этого сервера. В MySQL 5.5 реализована полусинхронная репликация, которая поддерживает этот метод естественным образом.)

Итоги главы

Правильное масштабирование MySQL немного менее гламурно, чем зачастую кажется. Правильный способ масштабирования — не строить будущую архитектуру Facebook с первого дня. Лучшая стратегия — делать то, что явно требуется для вашего приложения, и предполагать, что если оно на самом деле будет быстро расти, то ваши успехи позволят финансировать любые шаги, необходимые для решения новых задач.

Полезно иметь математическое определение масштабируемости и точную концепцию производительности, основываясь на законе универсальной масштабируемости. Знание того, что системы не могут масштабироваться линейно из-за последовательных инструкций и внутриузловой коммуникации, может помочь избежать возникновения этих проблем в приложении. В то же время многие проблемы масштабируемости не являются математическими, они могут быть вызваны проблемами внутри организации, такими как отсутствие командной работы или что-то менее конкретное. Книги *Guerrilla Capacity Planning* (автор — доктор Нил Дж. Гюнтер (Neil J. Gunther)) и *The Goal* (автор — Элияху М. Голдратт (Eliyahu M. Goldratt)) полезно прочитать всем, кто хочет понять, почему системы не масштабируются.

Что касается стратегий масштабируемости MySQL, то типичное приложение, которое сильно разрослось, обычно начинает с единственного сервера, а затем переходит на горизонтально масштабируемую архитектуру с чтением с подчиненных серверов. После этого приложение движется к шардированию и/или функциональному секционированию. Мы не согласны с теми, кто выступает за подход «шардируйте рано, шардируйте часто» для каждого приложения. Это сложно и дорого и многим приложениям никогда не понадобится. Ничто не мешает вам потратить время и посмотреть, что произойдет с новым оборудованием, новыми версиями MySQL или новыми разработками в MySQL Cluster, и даже оценить запатентованную систему, такую как Clustrix. В конце концов, шардирование — это ручная система кластеризации, так что не стоит изобретать велосипед.

При использовании нескольких серверов возникают проблемы с согласованностью и неделимостью. Наиболее распространенными затруднениями, с которыми мы сталкивались, были отсутствие согласованности сеансов (публикация комментария на сайте, обновление страницы и отсутствие комментария, который вы только что опубликовали) и сбои при передаче приложению информации о том, с каких серверов надо считывать данные и на какие записывать. Последние проблемы гораздо серьезнее, поскольку, если вы направляете записи более чем в одно место в своем приложении, то неизбежно столкнетесь с проблемами с данными, решать которые придется долго и мучительно. В этом могут помочь балансировщики нагрузки, однако они сами могут вызывать проблемы, иногда даже усугубляя те, которые призваны решить. Именно поэтому следующая глава посвящена высокой доступности.

12 Высокая доступность

В этой главе рассматривается последняя часть троицы, куда входят репликация, масштабируемость и высокая доступность. Под высокой доступностью мы подразумеваем малое время простоя. К сожалению, высокую доступность часто смешивают с близкими понятиями, такими как избыточность, защита от потери данных и балансировка нагрузки. Мы надеемся, что предыдущие две главы подготовили почву для ясного понимания высокой доступности. Однако нельзя сказать, что эта глава посвящена исключительно высокой доступности, — как и при разговоре о других частях троицы, в ней рассмотрены несколько связанных тем.

Что такое высокая доступность

На самом деле высокая доступность — это в чем-то мифический зверь. Обычно она выражается в процентах, что само по себе является подсказкой: не существует предельно высокой доступности, а лишь более или менее высокая доступность. Стопроцентная доступность просто недостижима. Правило девяток является наиболее распространенным способом выражения цели доступности. Как вы, возможно, знаете, пять девяток означает 99,999 % безотказной работы, что составляет чуть более 5 минут простоя в год. Это отлично для большинства приложений, хотя некоторые из них достигают еще большего числа девяток.

У разных приложений требования к доступности очень разнятся. Прежде чем формулировать, какую долю времени система должна быть доступной, подумайте, чего вы на самом деле хотите добиться. Каждое следующее увеличение степени доступности, как правило, обходится дороже предыдущего, отношение доступности к потраченным усилиям и затратам нелинейно. Рабочее время, в котором вы нуждаетесь, обычно зависит от того, что вы можете себе позволить. Вся идея с высокой доступностью заключается в уравновешивании стоимости простоя

с затратами на сокращение простоев. Иными словами, если вам нужно потратить много денег, чтобы увеличить время безотказной работы, но подобный результат позволит заработать лишь немногим больше, то, скорее всего, оно того не стоит. В общем, начиная с некоторого момента сделать приложение высокодоступным сложно и дорого, поэтому мы советуем ставить реалистичные цели и избегать чрезмерного усложнения. К счастью, усилия, необходимые для достижения двух или трех девяток времени безотказной работы, могут быть не очень высокими. Но это зависит от приложения.

Иногда люди определяют доступность как долю времени, в течение которого служба работает. Мы считаем, что это определение также должно включать в себя следующую характеристику: обслуживаются ли запросы с хорошей производительностью. Существует множество вариантов, при которых сервер может работать, но реально быть недоступным. Обычный случай — сразу после перезапуска сервера MySQL. Для большого сервера потребуется довольно много времени, чтобы разогреться и начать обслуживать запросы с приемлемым временем отклика, даже если сервер будет эксплуатироваться не на полную мощность.

Еще один фактор, который следует учитывать: возможна ли потеря информации, хотя приложение и не переставало работать. В случае катастрофического отказа сервера данные могут быть частично потеряны, по крайней мере те транзакции, которые были записаны в утраченный двоичный журнал, но не попали в журнал ретрансляции подчиненного сервера. С этим можно смириться? Для большинства приложений ответ положительный, если же это не так, то потребуется, как правило, дорогостоящее и сложное решение, сопряженное с дополнительными накладными затратами. Например, можно воспользоваться синхронной репликацией или поместить двоичный журнал на устройство, реплицируемое с помощью механизма DRBD, так что журнал не будет потерян даже при полном отказе системы. (Однако электричество может отключиться во всем центре хранения данных.)

Продуманная архитектура приложения зачастую может снизить требования к доступности, по крайней мере для части системы, так что обеспечить высокую доступность в целом будет проще. Разделение системы на критические и некритические части поможет сэкономить немало труда и денег, поскольку чем меньше система, тем легче обеспечить высокую доступность. Выявить наиболее значимые опасности можно, вычислив подверженность риску, то есть вероятность отказа, умноженную на стоимость этого отказа. Простенькая таблица рисков, содержащая столбцы «Вероятность», «Стоимость», «Подверженность риску», поможет распределить усилия по приоритетности.

В предыдущей главе мы рассмотрели способы достижения большей масштабируемости путем устранения причин плохой масштабируемости. Здесь мы используем аналогичный подход, поскольку считаем, что доступность проще всего понять, изучая ее противоположность — время простоя. Давайте начнем с обсуждения причин простоев.

Причины простоев

Нам доводилось слышать, что основной причиной простоев на серверах баз данных являются плохо написанные SQL-запросы. Но правда ли это? В 2009 году мы решили проанализировать нашу базу данных об инцидентах с клиентами и определить, какие причины на самом деле приводят к простоям и как их предотвратить[1]. Хотя ряд представлений был подтвержден полученными результатами, другие представления оказались ошибочными, а мы многому научились.

Сначала мы классифицировали случаи простоя не по причине, а по их проявлениям. Грубо говоря, ведущим местом проявления простоев оказалось то, что мы назвали операционной средой: здесь было зафиксировано около 35 % инцидентов. Операционная среда — это набор систем и ресурсов, которые поддерживают сервер базы данных, таких как операционная система, диски и сеть. Проблемы с производительностью заняли с небольшим отрывом второе место, здесь также было зафиксировано около 35 % инцидентов, связанных с простоями. Следующая — репликация, на которую приходится 20 % инцидентов, и оставшиеся 10 % пришлись на различные типы потери данных или повреждений, а также несколько иных проблем.

Классифицировав типы инцидентов, мы определили их причины. Приведем несколько основных.

- В операционной среде наиболее распространенной проблемой, лидирующей с большим отрывом, была нехватка дискового пространства.
- Самая распространенная причина простоя в группе проблем с производительностью — это действительно плохое выполнение SQL, хотя вина далеко не всегда ложилась на плохо написанные запросы. Например, множество проблем было вызвано ошибками сервера или его аномальным поведением.
- Плохие схема и проектирование индексов — следующие по распространенности проблемы производительности.
- Проблемы с репликацией обычно были вызваны различиями в данных между главным и подчиненными серверами.
- Проблемы с потерей данных обычно были вызваны командой `DROP TABLE` и всегда сопровождались отсутствием доступных резервных копий.

Обратите внимание на то, что репликация — одна из методик, направленных на увеличение времени безотказной работы, — часто приводит к простоям. Обычно это происходит из-за неправильного ее применения, так что данная ситуация иллюстрирует известный тезис: многие приемы, направленные на достижение высокой доступности, могут иметь обратные последствия. С этим мы столкнемся еще раз позже.

---

[1] Мы написали длинный технический документ, где провели полный анализ инцидентов, вызвавших простои у наших клиентов, а затем другой, в котором описали способы предотвращения простоев, включая подробные списки деятельности, которую нужно периодически проверять. В этой книге нет места для освещения всех этих подробностей, однако оба документа можно найти на сайте Percona (http://www.percona.com).

Теперь, когда известно, какие типы простоев встречаются и каковы причины их возникновения, можно подробно рассмотреть способы достижения высокой доступности.

Достижение высокой доступности

Высокая доступность достигается применением двух методик, которые должны идти рука об руку. Во-первых, попробуйте сократить время простоев, убрав их причины. Многие из них легко устраняются с помощью таких шагов, как правильная настройка, мониторинг и защита от человеческой ошибки. Во-вторых, постарайтесь обеспечить быстрое восстановление после простоев. Обычная тактика заключается в создании избыточности и отказоустойчивости в системах. Эти два уровня высокой доступности можно найти измерением двух показателей: среднего времени между отказами (mean time between failures, MTBF) и среднего времени восстановления (mean time to recovery, MTTR). Некоторые организации тщательно отслеживают их.

Второй пункт — быстрое восстановление за счет избыточности — к сожалению, кажется, привлекает наибольшее внимание, однако рентабельность инвестиций в усилия по предотвращению простоев может быть весьма высока. Давайте немного изучим мероприятия по их предотвращению.

Сокращение среднего времени между отказами

Вы можете избежать продолжительных простоев с помощью несложных мероприятий. Классифицируя случаи простоев и объясняя их причины, мы определили и способы их предотвращения. Мы обнаружили, что множество простоев может быть предотвращено просто с помощью разумного подхода к управлению системами. Следующие рекомендации выбраны из списка, приведенного в нашей работе, посвященной детальным результатам анализа.

- Протестируйте инструменты и процедуры восстановления, в том числе инструменты восстановления из резервных копий.
- Следуйте принципу минимизации привилегий.
- Держите свои системы в чистоте и порядке.
- Используйте качественные соглашения о наименованиях и организации. Это поможет избежать путаницы независимо от того, применяются ли эти серверы в разработке или в реальной работе.
- Составьте разумный график обновления сервера базы данных, тем самым поддерживая его актуальность.
- Тщательно тестируйте обновления перед их установкой, например, с помощью инструмента pt-upgrade из пакета Percona Toolkit.
- Используйте InnoDB, настройте ее правильно и убедитесь, что она является подсистемой хранения по умолчанию, а сервер не может запуститься, если она отключена.

- Убедитесь, что настройки основного сервера сконфигурированы правильно.
- Отключите DNS с помощью параметра `skip_name_resolve`.
- Если нужность кэша запросов не доказана, отключите его.
- Избегайте сложностей, таких как фильтры репликации и триггеры, если без них можно обойтись.
- Контролируйте важные компоненты и функции — особенно важные элементы, такие как дисковое пространство и состояние тома RAID. Но избегайте ложноположительных результатов — задавайте выдачу оповещений только о тех показателях, которые достоверно указывают на проблемы.
- Запишите как можно больше полученных ранее показателей, характеризующих состояние и производительность сервера, и сохраните их навсегда, если это возможно.
- Регулярно проверяйте целостность репликации.
- Назначьте подчиненным серверам режим «только для чтения» и не допускайте автоматического запуска репликации.
- Регулярно проверяйте запросы.
- Архивируйте и удаляйте ненужные данные.
- Зарезервируйте место в файловых системах. В GNU/Linux можете использовать параметр `-m` для резервирования места в самой файловой системе. Также можно оставить свободное пространство в группе томов LVM. Возможно, самый простой способ состоит в создании большого файла-заглушки, который можно будет удалить, если файловая система будет заполнена до конца[1].
- Приобретите привычку просматривать информацию об изменениях в системе, о ее состоянии и производительности, а также управлять ею.

Мы обнаружили, что провалы в управлении изменениями в системе — это наиболее важная причина инцидентов, связанных с простоями. Типичными ошибками являются неосторожные обновления, ошибки программы, вызванные отсутствием обновлений, изменение схемы или запроса для рабочей версии без их предварительного тестирования, отсутствие планирования таких моментов, как например, достижение предела емкости диска. Еще одна значимая причина проблем — отсутствие должной осмотрительности, например пренебрежение проверкой способности восстановления с резервных копий. Наконец, люди часто отслеживают в MySQL не то, что следовало бы. Получение сообщений о таких показателях, как, например, коэффициент попадания в кэш, которые не обозначают реальную проблему и выдают множество ложноположительных результатов, приводит к тому, что многие считают систему мониторинга бесполезной и игнорируют предупреждения. Иногда система мониторинга отказывает, и никто об этом даже не догадывается. А потом приходится отвечать на справедливые вопросы недовольного босса, например: «Почему Nagios не предупредила нас о том, что диск заполнен?»

[1] Это на 100 % кросс-платформенная совместимость!

Сокращение среднего времени восстановления

Как уже упоминалось, часто возникает соблазн, стремясь достичь высокой доступности, сосредоточиться исключительно на сокращении времени восстановления после простоев. Фактически иногда люди идут еще дальше и ограничиваются только одним аспектом сокращения времени восстановления, а именно предотвращением полного сбоя системы за счет создания в ней избыточности и устранения одиночных точек отказа.

Очень важно приложить усилия к достижению быстрого восстановления, ключевым аспектом чего является хорошая системная архитектура, обеспечивающая избыточность и отказоустойчивость. Однако получение высокой доступности — это не только техническая проблема. Необходимо принимать во внимание человеческий и организационный компоненты. Организации и отдельные лица различаются по уровню зрелости и способности избегать простоев и восстанавливаться после них.

Ваши сотрудники — важнейшие из ресурсов, необходимых для обеспечения высокой доступности, так что продуманные процедуры восстановления жизненно необходимы. Квалифицированные, способные адаптироваться, хорошо обученные сотрудники, вооруженные хорошо документированными и проверенными процедурами для работы в чрезвычайных ситуациях, могут в значительной степени способствовать быстрому восстановлению после простоев. Не стоит доверять исключительно инструментам и системам, поскольку они не понимают нюансов и временами делают, в принципе, правильные вещи, однако совершенно неприемлемые в данном конкретном случае.

Изучение инцидентов, связанных с простоями, может быть очень полезным с точки зрения обучения сотрудников и позволит избежать подобных инцидентов в будущем. Остерегайтесь, однако, переоценки таких методов, как «анализ постфактум», или «постмортем». Когда мы оглядываемся назад, реальность искажается, а стремление найти одну основную причину случившегося часто приводит к неверным выводам[1]. Многие популярные подходы, например «Пять "почему?"», не получается применить соответствующим образом, так как люди сосредоточены на поиске единственного козла отпущения. Трудно оглянуться назад на ситуацию, которая разрешилась, и понять ее реальные причины, тем более что их всегда много. Так что, хотя анализ постфактум может принести пользу, все же следует относиться к нему с изрядной долей скепсиса. Даже наши собственные рекомендации, основанные на длительном исследовании причин и способов предупреждения простоев, являются всего лишь частным мнением.

Стоит повторить: все случаи простоев происходят вследствие сочетания нескольких сбоев и, таким образом, для их предотвращения достаточно одного-единственного

[1] Два примера противоречий здравому смыслу можно встретить в следующих источниках: в статье Ричарда Кука (Richard Cook) How Complex Systems Fail (http://www.ctlab.org/documents/How%20Complex%20Systems%20Fail.pdf) и эссе Малькольма Гладуэлла (Malcolm Gladwell) о катастрофе шаттла «Челленджер», включенном в его книгу What the Dog Saw (издательство Little, Brown).

защитного механизма. Но необходимо предотвратить все эти сбои, а не разрушить в их цепи одно-единственное звено. Например, люди, которые обращаются к нам за помощью в восстановлении данных, обычно страдают как от потери данных (сбой работы хранилищ, ошибка администратора базы данных и т. п.), так и от отсутствия доступных резервных копий.

С учетом сказанного, нельзя говорить, что люди и организации проявляют излишнее рвение в расследовании причин простоев и попытках их предотвратить или ускорить восстановление после них. Вместо этого они часто сосредоточиваются на технических мерах, особенно крутых, таких как кластерные системы и избыточные архитектуры. В этом есть смысл, однако не забывайте, что системы также подвержены ошибкам. Один из инструментов, упомянутых во втором издании этой книги, а именно менеджер репликации MMM, больше не пользуется нашим расположением, поскольку оказалось, что он вызывает больше простоев, чем предотвращает. Вас, видимо, не удивит тот факт, что набор скриптов Perl иногда работает плохо, и даже чрезвычайно дорогие и сложные системы могут давать катастрофические сбои — да, даже SAN, которая стоит целое состояние. Мы видели множество сбоев SAN.

Устранение отдельных критических точек

Поиск и устранение отдельных критических точек в вашей системе в сочетании с механизмом переключения на использование запасного компонента — один из способов улучшения доступности за счет сокращения времени восстановления (MTTR). Хорошо постаравшись, можно сократить время восстановления до нуля, хотя это довольно сложно в общем случае. (Даже очень впечатляющие технологии, такие как дорогостоящие балансировки нагрузки, дают какую-то задержку — им нужно время, чтобы заметить проблемы и отреагировать на них.)

Подумайте о своем приложении и попытайтесь выделить все критические точки. Это может быть жесткий диск, сервер, коммутатор, маршрутизатор или питание для одного блока. Располагаются ли все ваши машины в одном ЦОДе или предоставлены ли резервные ЦОДы одной компанией?Любая точка в системе, которая не является избыточной, — это критическая точка. Обычными критическими точками в зависимости от ваших потребностей могут быть такие службы, как DNS, единственный сетевой провайдер[1], единственная облачная зона доступности и единственная энергосистема.

Далеко не всегда можно устранить отдельные критические точки. Создание избыточного компонента может оказаться невозможным из-за какого-то ограничения, с которым вы ничего не сможете сделать, например географического, бюджетного или временно́го. Попытайтесь учесть все компоненты, влияющие на доступность, составьте взвешенную картину рисков и начните работу с наиболее значимых. Неко-

[1] Это паранойя? Убедитесь, что ваши избыточные сетевые соединения действительно подключены к различным магистралям Интернета и что они физически не проложены по одной и той же улице или не висят на одном и том же столбе, поэтому не будут обрезаны одним и тем же экскаватором или оборваны одним автомобилем.

торые разработчики прилагают множество усилий к тому, чтобы создать программу, которая сможет пережить любые аппаратные сбои, но простои из-за ошибок в ней могут оказаться больше, чем если бы ее вообще не было. Те, кто строит «непотопляемые» системы, в которых резервировано все, что только можно, забывают, что центр обработки данных может быть обесточен или лишен доступа к сети. А быть может, разработчики не принимают во внимание разрушительные действия хакеров или неумышленные ошибки программистов, приведшие к удалению или повреждению данных. Случайная команда `DROP TABLE` тоже может стать причиной простоя.

Избыточность системы может принимать две формы: наличие запасной мощности и дублирование компонентов. Обеспечить запас мощности довольно просто — можно воспользоваться любым из упомянутых в этой и предыдущей главах методов. Один из способов повышения доступности состоит в том, чтобы создать кластер или пул серверов и применить тот или иной способ балансирования нагрузки. Если какой-нибудь сервер выходит из строя, то нагрузка перераспределяется между остальными. Вообще говоря, разумно по возможности не загружать компоненты под завязку, поскольку в этом случае остается запас для поддержания нужной производительности при возрастании нагрузки или отказе некоторых компонентов.

Дублирование компонентов может потребоваться во многих ситуациях. При этом придется держать резервный наготове, чтобы выполнить аварийное переключение на него в случае отказа основного. Дублировать можно сетевые карты, маршрутизаторы, накопители на жестких дисках — все, что с большой вероятностью способно выйти из строя. Дублировать сервер MySQL целиком несколько сложнее, поскольку без данных он бесполезен. Следовательно, нужно позаботиться о том, чтобы резервные серверы имели доступ к базам основного сервера. Разделяемое или реплицированное хранилище является одним из популярных способов достижения этого. Но является ли это архитектурой с высокой доступностью? Попробуем разобраться.

Разделяемое хранилище данных или реплицированный диск

Разделяемое хранилище — это способ ослабить связь между сервером базы данных и хранилищем данных. Обычно в этом качестве выступают SAN-сети. При устройстве такого хранилища активный сервер монтирует свою файловую систему и функционирует как обычно. Если активный сервер выходит из строя, то резервный может подмонтировать ту же самую файловую систему, выполнить операции восстановления и запустить MySQL для работы с файлами отказавшего сервера. Логически этот процесс ничем не отличается от починки отказавшего узла, только все происходит намного быстрее, так как резервный сервер уже загружен и готов к работе. Проверка файловой системы, восстановление на уровне InnoDB и «прогрев»[1] — самые большие

[1] Percona Server предлагает возможность восстановления сохраненного состояния буферного пула после перезапуска, и это отлично работает с разделяемым хранилищем. Тем самым время прогрева может сократиться на часы или даже дни. В MySQL 5.6 будет реализована аналогичная возможность.

задержки, с которыми вы, вероятно, столкнетесь при переходе на резервный сервер, однако в большинстве случаев само обнаружение сбоев может занять довольно длительное время.

У разделяемого хранилища есть два преимущества: оно помогает избежать потери данных при сбое любого другого компонента, кроме хранилища, а также позволяет создавать избыточность в компонентах, не связанных с хранением. Эта стратегия помогает снизить требования к доступности в некоторых частях системы, что упрощает достижение общей высокой доступности, поскольку усилия концентрируются на меньшем наборе компонентов. Но само разделяемое хранилище по-прежнему является единой точкой отказа. Если оно выходит из строя, вся система прекращает работу, и хотя SAN, как правило, очень хорошо спроектированы, они могут выйти из строя и выходят, иногда эффектно. Даже SAN, которые сами по себе являются избыточными, могут прекратить работу.

Доступ к разделяемому хранилищу в режиме «активный — активный»

Что можно сказать о запуске нескольких серверов в режиме «активный — активный» на SAN, NAS или в кластерной файловой системе? MySQL не может этого сделать. Она не предназначена для синхронизации доступа к данным с другими экземплярами MySQL, поэтому нельзя запустить несколько экземпляров MySQL, работающих с одними и теми же данными. (Технически вы могли бы, если бы использовали MyISAM для статических данных в режиме «только для чтения», но на практике мы никогда с таким не встречались.)

Подсистема хранения ScaleDB, работающая под MySQL, взаимодействует через API с архитектурой разделяемого хранилища, но мы не проверяли ее и не видели в реальных условиях. На момент написания книги она находится в стадии бета-тестирования.

Разделяемое хранилище подвержено определенным рискам. Если сбой, такой как, например, выход из строя MySQL, приводит к повреждению файлов данных, то резервный сервер может оказаться не в состоянии выполнить восстановление. Мы настоятельно рекомендуем использовать InnoDB или другую надежную ACID подсистему хранения с разделяемым хранилищем. Любой сбой почти наверняка повредит таблицы MyISAM, и их восстановление может занять много времени и привести к потере строк. Также настоятельно рекомендуем выполнять журналирование файловой системы с разделяемым хранилищем. Нам встречались серьезные невосстановимые повреждения файловой системой без журналирования и SAN (это была ошибка файловой системы, а не SAN).

Реплицируемый диск — еще один способ получить такой же результат, как и при использовании SAN. Совместно с MySQL чаще всего задействуется система DRBD (http://www.drbd.org) в сочетании с инструментами из проекта Linux-HA (подробнее об этом поговорим в дальнейшем).

DRBD — это синхронный механизм репликации блочного уровня, реализованный в виде модуля ядра Linux. Он копирует каждый блок первичного устройства по сети на другое блочное устройство (вторичное устройство) и записывает этот блок в место назначения еще до того, как он будет закоммичен на первичном устройстве[1]. Поскольку любая операция записи должна завершиться на вторичном устройстве раньше, чтобы запись на первичное считалась состоявшейся, то вторичное устройство должно быть как минимум таким же быстрым, как и первичное, иначе оно будет ограничивать производительность основной системы. Кроме того, если для создания резерва используется DRBD, то оборудование резервного сервера должно быть таким же, как и основного. Также при использовании DRBD необходим хороший RAID-контроллер с кэшем записи с резервным источником питания — без него производительность будет очень низкой.

Если активный сервер выходит из строя, можно сделать вторичное устройство первичным. Но поскольку DRBD реплицирует диск на блочном уровне, файловая система рискует оказаться рассогласованной. Следовательно, для быстрого восстановления лучше использовать журналируемую файловую систему. По завершении ее воссоздания MySQL, скорее всего, должна будет запустить собственные средства восстановления. Когда первый сервер снова вступит в строй, он синхронизирует свое устройство с новым первичным устройством и примет на себя роль вторичного.

С точки зрения фактической реализации переключения на резерв при отказе система DRBD аналогична SAN: имеется резервный компьютер, который начинает обслуживать те же данные, что и машина, вышедшая из строя. Основное отличие заключается в том, что это реплицируемая, а не разделяемая память, поэтому при использовании DRBD вы обслуживаете копию данных, тогда как в случае применения SAN речь идет об обслуживании той же самой информации, расположенной на том же физическом устройстве, к которому обращалась отказавшая машина. В обоих случаях кэши сервера MySQL будут пусты в тот момент, когда он запускается на резервной машине. Напротив, кэши подчиненного сервера репликации, скорее всего, будут хотя бы частично «прогреты».

У системы DRBD есть возможности, позволяющие предотвратить проблемы, свойственные ПО кластеризации. В качестве примера назовем *синдром разделения вычислительных мощностей*, который возникает, когда два узла пытаются сделаться первичными одновременно. DRBD можно сконфигурировать таким образом, что этот феномен никогда не возникнет. Однако система DRBD годится не для всех ситуаций. Рассмотрим некоторые присущие ей недостатки.

- Переключение на резервный сервер при сбое в DRBD осуществляется не мгновенно. Обычно для превращения вторичного устройства в первичное требуется несколько секунд, и это не считая восстановления файловой системы и MySQL.

[1] Уровень синхронизации DRBD поддается настройке. Можно установить асинхронный режим, режим, в котором система ждет получения данных удаленным устройством, или режим блокировки до завершения записи на удаленный диск. Кроме того, настоятельно рекомендуется отводить по DRBD отдельную сетевую карту.

- Работа DRBD весьма затратна, поскольку она функционирует только в активно-пассивном режиме. Когда реплицируемое устройство на сервере горячей замены работает в пассивном режиме, оно больше ни для чего не пригодно. Считать ли это недостатком, зависит от принятой точки зрения. Если требуется по-настоящему высокая доступность и вы не можете смириться с понижением уровня обслуживания при сбое, то ни на какую пару машин нельзя давать нагрузку, превышающую нагрузку на одну машину, поскольку в этом случае при выходе любой машины из строя оставшаяся не сможет справиться с работой. Конечно, можно использовать резервный сервер еще для чего-нибудь, например, как подчиненный сервер репликации, но часть ресурсов все равно будет простаивать.
- Для таблиц типа MyISAM этот механизм практически бесполезен, поскольку их проверка и восстановление занимают слишком много времени. Вообще, подсистема хранения MyISAM — неудачный выбор для обеспечения высокой доступности. Применяйте вместо нее InnoDB или любую другую подсистему с хорошей скоростью восстановления.
- Система DRBD не заменяет резервного копирования. Если данные на диске повреждены в результате злонамеренного вмешательства, ошибки, оплошности или аппаратного сбоя, то DRBD не поможет: реплицированные данные представляют собой точный дубликат испорченного оригинала. Для защиты от такого рода проблем необходима резервная копия (или репликация на уровне MySQL с задержкой по времени).
- Она добавляет некоторые накладные затраты для операции записи. Насколько они велики? Часто приводят значение в процентах, но это не очень хороший показатель. Вместо этого вам нужно понять, что операции записи страдают от дополнительной задержки из-за отправки и получения данных по сети и хранения на удаленном сервере, а эти затраты в относительном выражении значительнее для небольших записей. Хотя дополнительная задержка из-за сети может составлять всего около 0,3 с, что кажется сравнительно небольшим значением по сравнению с задержкой 4–10 мс при фактическом вводе/выводе на локальный диск, это примерно в три-четыре раза превышает задержку, которой можно ожидать от записи в кэш хорошего RAID-контроллера. Наиболее распространенной причиной замедления работы сервера с DRBD является то, что MySQL с InnoDB в режиме полной устойчивости выполняет много коротких операций записи и вызовов функции `fsync()`, которые сильно замедляются DRBD[1].

Мы предпочитаем использовать DRBD для репликации только тех устройств, на которых хранятся двоичные журналы. При выходе из строя активного сервера можно запустить на пассивном узле сервер журналов и использовать восстановленные двоичные журналы для приведения всех серверов, подчиненных отказавшему

[1] Большие последовательные записи — это совсем другая история. Дополнительная задержка из-за DRBD практически исчезает, но в игру вступают ограничения пропускной способности. Нормальный RAID-массив должен обеспечивать от 200 до 500 Мбайт в секунду пропускной способности последовательной записи, что намного выше того, чего может достичь сеть GigE.

главному, к последней точке синхронизации. После этого можно выбрать какой-то из подчиненных серверов и выдвинуть его на роль главного, заменив тем самым систему, вышедшую из строя.

В конечном счете разделяемые хранилища и реплицированные диски важны не столько для достижения высокой доступности (небольшого времени простоя), сколько для безопасного хранения данных. Пока у вас есть данные, вы можете восстановить данные после сбоев с показателем MTTR ниже, чем при невозможности сделать это (даже длительное время восстановления меньше, чем полное отсутствие восстановления). Однако по сравнению с архитектурами, которые позволяют ожидающему резервному серверу работать постоянно, у большинства разделяемых систем хранения или реплицированных дисков MTTR больше. Существует два способа поддержки сервера в постоянной готовности и запуска при необходимости: стандартная репликация MySQL, о которой мы говорили в главе 10, и синхронная репликация, которую обсудим в следующем разделе.

Синхронная репликация MySQL

При синхронной репликации транзакция на главном сервере не может завершиться, пока не будет закоммичена на одном или нескольких подчиненных. Тем самым достигаются две цели: никакие закоммиченные транзакции не теряются, если сервер выходит из строя, и есть по крайней мере еще один сервер с «живой» копией данных. Большинство синхронных архитектур репликации работают в активном режиме. Это означает, что каждый сервер является кандидатом на аварийное переключение в любое время, что упрощает большую доступность за счет избыточности.

На момент написания этой книги сама MySQL не предлагает синхронную репликацию[1], однако существуют два основанных на MySQL решения кластеризации, которые ее поддерживают. Кроме того, вы можете освежить в памяти главы 10, 11 и 13, в которых обсуждались другие интересные продукты, такие как Continuent Tungsten и Clustrix.

MySQL Cluster

Лучшим способом синхронной репликации в MySQL является MySQL Cluster (NDB Cluster). Он имеет синхронную репликацию в режиме «активный — активный» между всеми узлами. Это означает, что вы можете записывать на любой узел — все они одинаково способны читать и писать. Каждая строка хранится избыточно, поэтому вы можете потерять узел без потери данных, и кластер остается работоспособным. Хотя MySQL Cluster по-прежнему не является полным решением для всех типов приложений, как мы упоминали в предыдущей главе, в последних релизах он быстро улучшается. И теперь в нем реализован огромный список новых возможностей и особенностей: дисковое хранилище для неиндексированных данных, онлайновое масштабирование путем добавления узлов данных, таблицы `ndbinfo`

[1] Существует поддержка полусинхронной репликации в MySQL 5.5 (см. главу 10).

для управления кластером, скрипты для инициализации и управления кластером, многопоточные операции, перемещение операций соединения к узлам, содержащим данные, — сейчас это принято называть адаптивной локализацией запроса (adaptive query localization), способность обрабатывать типами BLOB и таблицами с большим количеством столбцов, централизованное управление пользователями и доступ к NoSQL через API NDB, а также протокол memcached. Предстоящие выпуски будут предусматривать возможность запуска в режиме согласованности в конечном счете с обнаружением и разрешением конфликтов транзакций в сети WAN для репликации между центрами данных в режиме «активный — активный». Короче говоря, MySQL Cluster — впечатляющий технологический продукт.

Существуют по крайней мере два поставщика настраиваемых продуктов для упрощения развертывания кластеров и управления ими: контракты на поддержку Oracle для кластера MySQL включают его продукт MySQL Cluster Manager, а Severalnines предлагает Cluster Control (http://www.severalnines.com). Этот продукт также может помочь в развертывании кластеров репликации и управлении ими.

Percona XtraDB Cluster

Percona XtraDB Cluster — относительно новая технология, которая добавляет возможность синхронной репликации и кластеризации самой подсистеме хранения XtraDB (InnoDB), а не через новую подсистему хранения или внешний сервер. Он построен на Galera[1] — библиотеке, которая реплицирует записи по узлам в кластере. Как и MySQL Cluster, Percona XtraDB Cluster предлагает синхронную репликацию с несколькими главными серверами[2] с возможностью записи на любом узле. Как и MySQL Cluster, он может обеспечить высокую доступность, а также гарантировать нулевую потерю данных (устойчивость — D (durability) в ACID), когда узел выходит из строя. И конечно же, узлы могут выйти из строя, не вызывая сбоя всего кластера.

В лежащей в основе сервера технологии Galera используется методика, называемая *репликацией множества строк, изменяемых при записи.* Изменяемое множество строк кодируется как построчные события двоичного журнала для передачи их между узлами и обновления других узлов в кластере, хотя двоичный журнал включать не обязательно.

Percona XtraDB Cluster работает очень быстро. Репликация между узлами может быть более быстрой, чем некластеризованная, поскольку запись в удаленную оперативную память выполняется быстрее, чем запись на локальный диск в режиме

---

[1] Технология Galera разработана Codership Oy (http://www.codership.com) и доступна в виде патча для стандартных MySQL и InnoDB. Percona XtraDB Cluster включает в себя модифицированную версию этого набора патчей, а также другие возможности и функции. Percona XtraDB Cluster — это решение на базе Galera, поставляемое в готовом виде.

[2] Вы также можете использовать его в конфигурации «главный сервер — подчиненный сервер», выполняя операции записи только на один узел, но при этом режиме работы в конфигурации кластера нет никаких различий.

полной устойчивости. Если есть желание, можно повысить производительность за счет снижения устойчивости на каждом узле, при этом поддерживая устойчивость за счет наличия нескольких узлов с копиями данных. NDB работает по такому же принципу. Устойчивость кластера в целом не снижается — уменьшается только локальная устойчивость. Кроме того, он поддерживает построчную параллельную (многопоточную) репликацию, поэтому для выполнения изменения множества строк можно использовать несколько ядер процессора. Эти особенности в совокупности делают Percona XtraDB Cluster привлекательным в средах облачных вычислений, где диски и процессоры, как правило, медленнее обычных.

Для того чтобы узлы не генерировали конфликтующие значения, кластер реализует автоинкрементные ключи с помощью параметров `auto_increment_offset` и `auto_increment_increment`. Блокировка обычно производится так же, как и в стандартной InnoDB. Используется оптимистический контроль конкурентности. Изменения сериализуются и передаются между узлами после коммита транзакции с процессом сертификации, так что, если возникают конфликты при обновлении, кто-то должен уступить. Если несколько узлов одновременно меняют одни и те же строки, это может привести к множеству взаимных блокировок и откатов.

Percona XtraDB Cluster обеспечивает высокую доступность, сохраняя узлы онлайн, пока они образуют кворум. Если узлы обнаруживают, что они не являются частью кворума, они отделяются от кластера, как следствие им приходится повторно выполнять синхронизацию, прежде чем снова присоединиться к кластеру. В результате кластер не может обрабатывать сценарии разделения вычислительных мощностей: если такой сценарий встретится, работа прекратится. Если в двухузловом кластере выходит из строя один узел, оставшийся не является кворумом и перестает функционировать, поэтому на практике для кластера высокой доступности требуется использовать как минимум три узла.

Percona XtraDB Cluster обладает множеством преимуществ.

- Он обеспечивает прозрачную кластеризацию на основе InnoDB, поэтому нет необходимости переходить на другую технологию, например на NDB, которая разительно отличается с точки зрения обучения и администрирования.
- Он на самом деле обеспечивает высокую доступность, при этом все узлы равны и готовы к чтению и записи в любое время. Напротив, встроенная в MySQL асинхронная или полусинхронная репликация требует наличия только одного главного сервера. Тем самым не гарантируется репликация ваших данных, а также актуальность подчиненных серверов и их готовность выполнить чтение или быть повышенными до главного сервера.
- Он защищает от потери данных при сбое узла. Фактически, поскольку все узлы содержат все данные, вы можете позволить выйти из строя всем узлам, кроме одного, и все еще не потерять данные, даже если в кластере применяется разделение вычислительных мощностей и он перестал работать. Этим он отличается от NDB, где данные секционированы на группах узлов и некоторые данные могут быть потеряны, если все серверы группы выйдут из строя.

- Подчиненные серверы не могут отстать от главного сервера, поскольку изменения множества строк распространяются и сертифицируются на каждом узле кластера до коммита транзакции.
- Он использует построчный журнал событий для применения изменений в подчиненных серверах, поэтому изменение множества строк может быть менее затратным, чем их создание, как и при обычной построчной репликации. Это в сочетании с многопоточным применением наборов записей может сделать его репликацию более масштабируемой, чем репликация MySQL.

Конечно же, мы обязаны упомянуть об его недостатках.

- Он новый, опыт его использования сравнительно невелик. Поэтому нет подробной информации о его сильных и слабых сторонах, а также о подходящих вариантах использования.
- Весь кластер выполняет запись так же медленно, как самый медленный узел. Таким образом, для всех узлов необходимо примерно одинаковое аппаратное обеспечение, и если один узел замедляется (например, поскольку RAID-карта выполняет цикл проверки батареи), все остальные тоже замедляются. Если вероятность медленного выполнения одним узлом равна P, трехузловой кластер будет медленным с вероятностью $3P$.
- Он не так эффективно использует пространство, как NDB, поскольку каждый узел хранит все данные, а не только их часть. В то же время он основан на Percona XtraDB, которая является усовершенствованной версией InnoDB, поэтому у него, в отличие NDB, нет ограничений в отношении данных на диске.
- В настоящее время он запрещает применение некоторых операционных приемов, возможных при асинхронной репликации. Например, таких, как изменение схемы в автономном режиме на подчиненном сервере и повышение его до уровня главного сервера, чтобы можно было повторить в автономном режиме изменения на других узлах. Альтернативой является использование такой методики, как онлайновый инструмент преобразования схем пакета Percona Toolkit.
- При добавлении нового узла в кластер необходимо скопировать на него данные. Кроме того, он должен быть синхронизирован с текущими операциями записи, поэтому большой кластер с большим количеством операций записи трудно наращивать. Это накладывает ограничения на размер данных кластера. Мы не знаем, каким должен быть этот размер, но, согласно пессимистической оценке, ограничения начнут сказываться уже примерно при 100 Гбайт. Хотя критический размер может оказаться и бóльшим — время и опыт покажут.
- На момент написания книги протокол репликации, по-видимому, довольно чувствителен к небольшим сетевым сбоям, что может привести к остановке узлов и выходу их из кластера, поэтому рекомендуем использовать высокопроизводительную сеть с хорошей избыточностью. Если у вас нет надежной сети, скорее всего, вам придется постоянно возвращать узлы обратно в кластер. Для этого требуется повторная синхронизация данных. На момент написания книги инкрементная передача состояния, позволяющая избежать копирования полного набора

данных, почти готова к использованию, так что в будущем это, по-видимому, перестанет быть проблемой. Кроме того, можно сконфигурировать Galera так, чтобы она была более терпимой к сетевым задержкам (ценой отложенного обнаружения сбоев), а для будущих релизов запланированы более надежные алгоритмы.

- Если вы не будете тщательно следить за кластером, он может стать слишком большим и не сможет перезапускать выходящие из строя узлы, так же как и резервные копии могут стать слишком большими для восстановления в разумные сроки. У нас не хватает практического опыта, чтобы точно сказать, как это будет работать на практике.
- Из-за взаимодействия между узлами, необходимого при коммите транзакции, по мере добавления узлов в кластер операции записи будут замедляться, а взаимоблокировки и откаты — становиться все более частыми. (Более подробную информацию о том, почему это происходит, см. в предыдущей главе.)

Как Percona XtraDB Cluster, так и Galera все еще находятся на раннем этапе своего жизненного цикла, быстро меняются и постоянно совершенствуются. Сейчас мы можем сказать о недавних или предстоящих улучшениях поведения кворума, безопасности, синхронности, управления памятью, передачи состояния и множества других элементов. Кроме того, в будущем вы сможете использовать узлы в автономном режиме для таких операций, как локальное изменение схемы.

Избыточность на основе репликации

Менеджеры репликации — это инструменты, которые пытаются использовать стандартную репликацию MySQL как строительный блок для избыточности[1]. Хотя есть возможность улучшить доступность с помощью репликации, существует «стеклянный потолок», который не позволяет текущей асинхронной и полусинхронной репликации MySQL получить те же результаты, которые достигаются с помощью истинной синхронной репликации. Вы не можете гарантировать мгновенный переход на другой ресурс и нулевую потерю информации, а также не имеете права рассматривать все узлы как идентичные.

Менеджеры репликации обычно контролируют три вещи: коммуникацию между приложением и MySQL, работоспособность серверов MySQL и отношения репликации между серверами MySQL, а также управляют ими. Они либо изменяют конфигурацию балансировки нагрузки, либо по мере необходимости перемещают виртуальные IP-адреса, чтобы приложение подключалось к надлежащим серверам, а также управляют репликацией, выбирая сервер, на который происходит запись в псевдокластере. В принципе, это несложно: просто убедитесь, что записи никогда не направляются на сервер, который не готов к записи, и получите правильные координаты репликации при повышении подчиненного сервера до главного.

---

[1] В этом разделе мы стараемся избежать путаницы. Избыточность — это не то же самое, что высокая доступность.

Это приемлемо звучит в теории, однако опыт показывает, что на практике все так хорошо работает далеко не всегда. На самом деле это очень плохо, поскольку иногда было бы хорошо иметь облегченный набор инструментов, помогающих восстановлению при обычных сбоях, и получить немного более высокую доступность без больших затрат. К сожалению, мы не знаем ни одного хорошего набора инструментов, который надежно решает эту задачу. Назовем два менеджера репликации[1], однако один из них — новый, а в другом — масса проблем.

Нам доводилось видеть, как многие пытаются написать свои менеджеры репликации. При этом они обычно попадают в те же ловушки, что и другие до них. Не стоит создавать собственные продукты. В этом случае возникает риск потери данных, поскольку вы, скорее всего, к нормально работающим асинхронным компонентам добавите ошибочные режимы. Если вы никогда не сталкивались с этими ошибками, вам будет очень сложно написать программу так, чтобы она должным образом обрабатывала их. Фактически машина может попасть в ситуацию, которую способен исправить квалифицированный пользователь и которую можно усугубить, совершая неправильные действия.

Первый менеджер репликации, который мы хотим назвать, — MMM (http://mysql-mmm.org). У авторов этой книги нет единого мнения о том, насколько подходит этот набор инструментов для развертывания в производственных условиях (хотя его автор предположил, что он не заслуживает доверия). Кое-кто из нас считает, что в некоторых случаях этот инструмент может быть полезен, если его применять в режиме ручного аварийного переключения, другие предпочли бы никогда не использовать его ни в каком режиме. Однако множество наших клиентов, у которых он работает в режиме автоматического аварийного переключения, имеют с ним множество серьезных проблем. Он может выводить офлайн работоспособные серверы, отправлять записи в неправильное место и задавать подчиненным серверам неправильные координаты. Иногда это приводит к хаосу.

Другим средством, довольно новым, является инструментарий MHA Йошинори Мацунобу (Yoshinori Matsunobu) (http://code.google.com/p/mysql-master-ha/). Он похож на MMM в том, что это набор сценариев для создания псевдокластера, действующий схожими методами. Однако это не полная замена: он не пытается выполнять столько же действий и полагается на Pacemaker при перемещении виртуальных IP-адресов. Одно из главных отличий заключается в том, что у него есть набор тестов, способный предотвратить часть проблем, с которыми столкнулся MMM. Пока мы не сформировали обоснованного мнения об этом инструменте. Мы не используем его сами и не обсуждали ни с кем, кроме Йошинори, который применяет его в реальных условиях.

Избыточность на основе репликации — это в конечном счете палка о двух концах. Ее целесообразно использовать тогда, когда доступность гораздо важнее, чем согласованность или гарантии отсутствия потерь данных. Например, некоторые люди

[1] Сейчас мы работаем над решением, основанным на Pacemaker и стеке Linux-HA, но пока не готовы назвать его.

в действительности зарабатывают деньги не функционалом своего сайта, а только его доступностью. Кого волнует, что случился сбой и с сайта пропали несколько комментариев, фотографии или что-то еще? Пока доход от рекламы продолжает расти, затраты на обеспечение действительно высокой доступности могут не окупаться. Однако тактика приложения максимальных усилий для поддержания высокой доступности с помощью репликации, несет в себе потенциальную угрозу серьезных простоев, восстановление после которых может оказаться чрезвычайно трудоемким занятием. Во многом это рулетка, вероятно, слишком рискованная для всех, кроме самых блаженных (или опытных) пользователей.

Проблема заключается в том, что многие не знают, как самостоятельно оценить, подходит ли для них рулетка репликации. Этому есть две причины. Во-первых, они не видят «стеклянный потолок» и ошибочно полагают, что набор виртуальных IP-адресов, репликации и скриптов управления может обеспечить реально высокую доступность. Во-вторых, недооценивают сложность технологии и, следовательно, серьезность сбоев, которые могут произойти, и трудность восстановления после них. В результате иногда люди считают, что могут справиться с избыточностью на основе репликации, но позже, возможно, жалеют, что не выбрали более простую систему, обеспечивающую большие гарантии.

У других типов репликации, таких как DRBD или SAN, есть свои недостатки — пожалуйста, не думайте, что мы продвигаем их как железобетонные и говорим, что репликация MySQL плоха, — у нас даже в мыслях такого нет. Написать плохие скрипты для аварийного переключения для DRBD так же легко, как и для репликации MySQL. Основное различие заключается в том, что репликация MySQL намного сложнее, с гораздо большим количеством нюансов, и она не мешает вам совершать ошибки.

Аварийное переключение и восстановление после сбоев

Избыточность — это здорово, но на самом деле она не дает ничего, кроме возможности оправиться от сбоя. (Ха-ха, вы можете добиться этого с помощью резервных копий.) Избыточность не увеличивает доступность и не сокращает время простоев ни на йоту. Высокая доступность построена поверх избыточности с помощью процесса аварийного переключения. При наличии избыточности в случае выхода компонента из строя вы можете прекратить его использование и начать использовать его избыточный резерв, находящийся в состоянии ожидания. Сочетание избыточности и аварийного переключения позволяет вам быстрее восстанавливаться, а как вы знаете, уменьшение MTTR сокращает время простоев и повышает доступность.

Мы последовательно употребляем здесь термин «обход/обработка отказа» (failover), некоторые используют в качестве синонима термин «аварийный возврат к первичному серверу» (fallback). Также говорят о «переключении на резервный сервер», чтобы подчеркнуть, что речь идет о запланированном переключении, а не о реакции на аварийную ситуацию. Конечно, конечно, все именно так и было.

Мы используем термин «восстановление после сбоя» (failback) для обозначения процедуры, обратной переключению на резервный сервер. Если в систему встроен механизм возврата на основной сервер, то переключение на резервный становится двухэтапным процессом: после того как сервер А отказал и был заменен сервером Б, можно починить сервер А и вернуть ему роль основного.

Аварийное переключение подходит не только для восстановления после сбоев. Вы также можете планировать аварийное переключение для сокращения простоев (улучшения доступности) во время таких событий, как обновление, изменение схемы, модификация приложения или плановое техническое обслуживание.

Нужно определить, как быстро должно выполняться аварийное переключение, также следует знать, как скоро после этого вы должны заменить неисправный компонент. До тех пор пока вы не восстановите исчерпанную резервную мощность системы, ее избыточность будет меньше, а значит, она подвергается повышенному риску. Таким образом, наличие ожидающего сервера не устраняет необходимости своевременной замены неисправных компонентов. Как быстро вы сможете создать новый ожидающий сервер, установить на него операционную систему и загрузить актуальную копию данных? У вас достаточно ожидающих машин? Возможно, понадобится больше одной.

Есть много разновидностей аварийного переключения на резервный сервер при отказе. Некоторые мы уже обсуждали, поскольку во многих отношениях балансирование нагрузки и аварийное переключение схожи и граница между ними размыта. В целом мы полагаем, что полноценное предназначенное для обхода отказов решение как минимум должно обеспечивать мониторинг и автоматическую замену компонентов. В идеале все это должно быть прозрачно для приложения. Балансирование нагрузки такими возможностями не располагает.

В мире UNIX для аварийного переключения обычно используются инструменты, разрабатываемые в рамках проекта High Availability Linux (http://linux-ha.org), которые работают во многих UNIX-подобных операционных системах. За последние несколько лет стек Linux-HA стал значительно более функциональным. Сегодня большинство людей считает Pacemaker основным компонентом в стеке. Pacemaker заменил старый инструмент пульсации. Прочие инструменты обладают функциями захвата IP и балансировки нагрузки. Все это можно сочетать с DRBD и/или LVS

Самая важная часть механизма аварийного переключения — это восстановление. Если невозможно по желанию менять роли серверов, то аварийное переключение — тупик, который только отсрочивает останов. Поэтому нам так нравятся симметричные топологии репликации, например конфигурация с двумя главными серверами, и не нравится топология «кольцо», когда главных серверов три или больше. Если конфигурация симметрична, то переключение и возврат — одна и та же операция, выполняемая в разных направлениях (стоит отметить, что в систему DRBD встроены средства возврата).

Для некоторых приложений критически важно, чтобы переключение и возврат управления производились настолько быстро и одномоментно, насколько возможно. Но даже если это не так, все равно не стоит полагаться на то, что вам неподконтрольно,

например на изменение параметров DNS или конфигурационных файлов приложения. Некоторые наиболее серьезные проблемы не проявляются, пока система не станет достаточно велика и такие неприятности, как вынужденный перезапуск приложений или необходимость в единовременной операции, не встанут во весь рост.

Поскольку балансирование нагрузки и переключение при отказе так тесно связаны и часто для решения обеих задач применяется одно и то же аппаратное и программное обеспечение, мы рекомендуем выбирать такую методику балансирования нагрузки, которая поддерживала бы и переключение. Именно по этой причине не советуем выполнять балансирование нагрузки с помощью изменения DNS или кода. При подобном подходе вы только создаете себе лишнюю работу: существующий код придется переписать, когда потребуется обеспечить высокую доступность.

В следующих разделах обсуждаются некоторые распространенные методы аварийного переключения.

Повышение подчиненного сервера, или перемена ролей

Повышение подчиненного сервера до уровня главного, или перемена ролей (активной и пассивной), в топологии репликации «главный — главный» — важная составная часть многих решений по переключению на резерв при отказе для MySQL. О том, как это сделать в ручном режиме, подробно написано в главе 10. А ранее в этой главе упоминалось, что мы не знаем никаких автоматизированных инструментов, которые правильно действуют во всех ситуациях, или по крайней мере ни одного, за который мы могли бы поручиться репутацией.

Наличие рабочей нагрузки не позволит вам мгновенно перейти к пассивному подчиненному серверу. Подчиненные серверы воспроизводят записи главного, но если при этом не используются для чтения, то будут недостаточно «разогреты» для того, чтобы обслуживать производственную нагрузку. Если вы хотите, чтобы подчиненный сервер был готов к чтению трафика, то должны постоянно «обучать» его, либо позволяя находиться под рабочей нагрузкой, либо зеркалируя на него реальные запросы на чтение. Мы иногда делали так: анализировали трафик TCP, отфильтровывали все, кроме запросов `SELECT`, и воспроизводили их на подчиненном сервере. В Percona Toolkit есть инструменты, которые могут помочь с этим.

Виртуальные IP-адреса и передача IP-адреса

Можно назначить логический IP-адрес экземпляру MySQL, который должен выполнять определенные функции. Если этот экземпляр выйдет из строя, IP-адрес можно будет передать другому серверу MySQL. По существу, это тот же самый подход, который был описан в предыдущей главе, только теперь он используется для переключения, а не для балансирования нагрузки.

Достоинство этого подхода в его прозрачности для приложения. Конечно, существующие соединения будут разорваны, но изменять конфигурацию ПО не придется. Кроме того, передачу IP-адреса можно осуществить одномоментно, так что все

приложения увидят результат изменения одновременно. Это особенно важно, когда сервер раскачивается между доступным и недоступным состояниями.

Недостатки этого процесса следующие.

- Либо все IP-адреса должны находиться в одном и том же сегменте сети, либо необходимо организовать соединение сетей типа «мост».
- Для изменения IP-адреса требуется доступ в систему с правами администратора.
- Иногда приходится обновлять кэши протокола разрешения адресов (ARP). Некоторые сетевые устройства хранят записи в ARP-кэше слишком долго, поэтому соответствие между IP-адресом и MAC-адресом компьютеров изменяется не мгновенно. Нам встречалось множество случаев, когда сетевое оборудование или другие компоненты не взаимодействуют и поэтому различные части системы не понимают, где действительно хранится IP-адрес.
- Сетевое оборудование должно поддерживать передачу IP-адреса. Иногда для правильной работы требуется иметь возможность клонировать MAC-адреса.
- Бывает, что сервер сохраняет за собой IP-адрес, хотя и не функционирует в полном объеме. Тогда необходимо физически останавливать сервер и отключать его от сети. Это действие известно под прекрасным акронимом STONITH: «Стрелять в другой узел в голове» (Shoot the other node in the head). Также это действие называют фехтованием — это название элегантнее и звучит как официальное.

Плавающие IP-адреса и передача адреса от одного компьютера другому хорошо работают, когда нужно обеспечить перехват управления машиной, которая расположена рядом с отказавшей, то есть в той же самой подсети. Однако вы должны понимать, что эта стратегия не всегда железобетонна, поскольку ее результат зависит от сетевого оборудования и т. д.

Ожидание распространения изменений

Часто при организации избыточности на некотором уровне приходится ждать, пока нижележащий уровень выполнит изменение. Ранее в этой главе мы уже отмечали, что замена серверов средствами DNS — плохое решение, так как изменения в системе DNS распространяются слишком медленно. Замена IP-адресов дает больший контроль, но распространение IP-адресов в локальной сети зависит от уровня, который находится еще ниже, — протокола ARP.

Решения на основе посредника

Для реализации переключения на резерв и возврата на основной сервер при отказе можно использовать прокси-серверы, переадресацию портов, трансляцию сетевых адресов (NAT) и аппаратные балансировщики нагрузки. Эти решения хороши, поскольку, в отличие от других, которые часто создают неопределенность (все ли

компоненты системы действительно согласны с тем, какая из них является главной базой данных? Может ли она быть изменена мгновенно и одним действием?), они выступают как центральный орган, контролирующий соединения между приложением и базой данных. Однако они сами становятся едиными точками отказа, поэтому во избежание проблем необходимо делать их избыточными.

Среди прочего с помощью такого рода решения можно сделать так, что удаленный центр обработки данных будет выглядеть как находящийся в той же сети, что и приложение. Это позволяет использовать такие методы, как плавающие IP-адреса, чтобы приложение начало взаимодействовать с совершенно другим центром обработки данных. Можно сконфигурировать сервер приложений в каждом центре обработки данных так, чтобы он подключался к собственному посреднику, который маршрутизирует трафик компьютерам в активном центре обработки данных. Подобная конфигурация изображена на рис. 12.1.

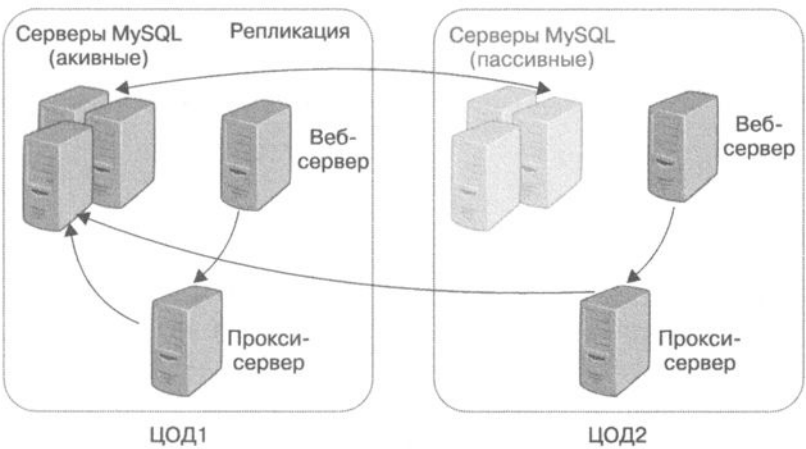


Рис. 12.1. Использование MySQL посредника для маршрутизации соединений между центрами обработки данных

Если активный центр обработки данных полностью выходит из строя, то посредник может начать маршрутизировать трафик пулу серверов в другом центре, и приложению знать об этом ни к чему. Основной недостаток такой конфигурации — большая задержка между сервером Apache в одном центре обработки данных и серверами MySQL в другом. Чтобы сгладить эту проблему, можно запускать веб-сервер в режиме переадресации. В этом случае весь трафик перенаправляется в центр обработки данных, где находится пул активных серверов MySQL. Того же эффекта можно добиться с помощью HTTP-прокси.

На рис. 12.1 средством соединения серверов MySQL является программа MySQL Proxy, но подойдут и другие архитектуры с посредником, например LVS и аппаратные балансировщики нагрузки.

Обработка переключения на резерв при отказе на уровне приложения

Иногда более простым или гибким действием является обработка переключения на резерв при отказе на уровне самого приложения. Например, если в приложении возникает ошибка, которая обычно не обнаруживается внешним наблюдателем, скажем, поступает сообщение, свидетельствующее о повреждении данных, то оно может само запустить процедуру переключения.

Хотя идея интеграции процесса переключения при отказе может показаться привлекательной, на самом деле этот подход не очень удачен. Большинство приложений состоит из многих компонентов: заданий cron, конфигурационных файлов, скриптов на разных языках программирования. Поэтому добавление сюда еще и переключения делает приложение излишне громоздким, особенно по мере его естественного роста и усложнения.

Однако встроить в ПО средство мониторинга и позволить ему инициировать процесс переключения — вполне здравая идея. Кроме того, приложение должно уведомлять пользователей о снижении функциональности с помощью соответствующих сообщений.

Итоги главы

Высокой доступности можно достичь за счет сокращения времени простоев, на которое следует воздействовать с двух сторон: увеличивая время между отказами (MTBF) и уменьшая время восстановления после сбоев (MTTR).

Чтобы увеличить время между отказами, попытайтесь их предотвратить. К сожалению, когда вы предотвращаете отказы, многим кажется, что вы ничем не занимаетесь, так как профилактическая работа часто не учитывается. Мы назвали основные моменты процесса предотвращения простоев в системах MySQL (скучные детали рассматриваются в наших статьях, доступных по адресу http://www.percona.com). Старайтесь учиться на своих простоях, но остерегайтесь возводить на пьедестал анализ основных причин и постмортем.

Сокращение времени восстановления может стать сложным и затратным. В то же время, выполняя мониторинг, вы можете рано обнаружить проблемы и записать значения множества показателей, что поможет диагностировать проблемы. Кроме того, эти показатели иногда можно использовать для определения проблем до того, как они приведут к простоям. Наблюдайте и выдавайте оповещения избирательно, чтобы они не превращались в шум, но при этом тщательно записывайте показатели состояния и производительности.

Другим приемом сокращения времени восстановления является создание избыточности в системе и обеспечение возможности аварийного переключения так, чтобы

вы могли переключаться между избыточными компонентами, когда один из них выходит из строя. К сожалению, избыточность делает системы очень сложными. Теперь основные процессы не централизованы — они распределены, а это означает необходимость координации и синхронизации, а также применения теоремы CAP, задачи византийских генералов и пр. Вот почему системы наподобие NDB Cluster трудно как просто создать, так и сделать достаточно общими, чтобы можно было обслуживать рабочие нагрузки любого пользователя. Но ситуация постепенно улучшается, и, возможно, в четвертом издании мы сможем спеть осанну одной или нескольким кластерным базам данных.

В этой и двух предыдущих главах рассмотрены темы, которые часто объединяют: репликация, масштабируемость и высокая доступность. Мы попытались рассмотреть их по отдельности, поскольку полезно четко их разделять. Итак, как связаны эти три главы?

Люди обычно хотят получить от своих баз данных по мере роста их приложений три возможности:

- возможность добавлять мощность для обслуживания увеличенной нагрузки без ущерба для производительности;
- защиту от потери закоммиченной транзакции;
- чтобы приложения оставались онлайновыми и обслуживали транзакции, тем самым продолжая зарабатывать деньги.

Достижение этого комплекса целей обычно начинают с избыточности. Она в сочетании с механизмом аварийного переключения обеспечивает высокую доступность за счет минимизации MTTR. Кроме того, избыточность добавляет резервные мощности для обслуживания большей нагрузки.

Разумеется, нужно дублировать и данные, а не только ресурсы. Это может помочь предотвратить потерю данных при выходе из строя сервера, что добавляет устойчивости. Единственный способ дублирования данных — каким-то образом их реплицировать. К сожалению, при дублировании данных возникает вероятность несогласованности, а тем самым и необходимость в координации и коммуникации между узлами. Из-за этого в системе увеличиваются накладные затраты; поэтому системы являются более или менее масштабируемыми.

Дублирование также требует больше ресурсов (больше жестких дисков, больше оперативной памяти и т. д.), что увеличивает затраты. Один из способов снизить как потребление ресурсов, так и накладные затраты на поддержание согласованности заключается в секционировании (шардировании) данных и распределении каждого шарда на определенную систему. Это уменьшает дублирование данных и ослабляет зависимость избыточности данных из избыточности ресурсов.

Итак, слово за слово, мы возвращаемся к группе связанных концепций и приемов для достижения целого набора целей. Однако это не просто разные способы говорить об одном и том же.

В конце концов, нужно выбрать стратегию, которая подходит вам и вашему приложению. Решение о полной сквозной стратегии высокой доступности не следует принимать на базе простых эмпирических правил. Возможно, мы можем помочь, предложив приблизительные рекомендации.

Чтобы добиться очень небольшого времени простоев, вам понадобятся избыточные серверы, которые готовы мгновенно принять на себя рабочую нагрузку приложения. Они должны быть онлайн и постоянно обрабатывать запросы, а не просто ожидать, то есть должны быть «разогреты» и готовы к работе. Если вам нужны твердые гарантии, то подойдет кластерный продукт, такой как MySQL Cluster, Percona XtraDB Cluster или Clustrix. Если небольшое проседание производительности в процессе аварийного переключения некритично, то хорошей альтернативой может быть стандартная репликация. Будьте осторожны при использовании автоматических механизмов аварийного переключения — в определенных ситуациях они могут уничтожить данные.

Если для вас время аварийного переключения не очень значимо, но нужно избежать потери данных, то вам потребуется строго гарантированная избыточность данных, то есть синхронная репликация. На уровне хранилища обеспечить ее с минимальными затратами поможет DRBD, на другом конце спектра затрат можете использовать две SAN, синхронно реплицированные между собой. Кроме того, вы можете реплицировать данные на уровне базы данных, используя такие технологии, как MySQL Cluster, Percona XtraDB Cluster или Clustrix. А еще можно задействовать промежуточное программное обеспечение, такое как Tungsten Replicator. Если вам не нужна сильная защита и вы хотите, чтобы все оставалось как можно более простым, обычная асинхронная или полусинхронная репликация может стать хорошим вариантом по разумной цене.

Либо можно просто разместить приложение в облаке. Почему нет? Разве это не сделает его доступным и бесконечно масштабируемым? Давайте выясним.

13 MySQL в облаке

Множество людей используют MySQL в облаке, иногда в очень больших масштабах, и это неудивительно. По нашим наблюдениям, большинство из них задействуют платформу Amazon Web Services (AWS), в частности тома Amazon Elastic Compute Cloud (EC2), Elastic Block Store (EBS) и в меньшей степени Relational Database Service (RDS).

Один из способов начать обсуждение MySQL в облаке — разделить ее на две большие категории.

- *IaaS (Infrastructure as a Service — инфраструктура как услуга).* IaaS — это облачная инфраструктура для размещения вашего собственного сервера MySQL. Вы можете приобрести ресурс виртуального сервера в облаке и использовать его для установки и запуска своего экземпляра MySQL. У вас есть возможность конфигурировать MySQL и операционную систему по своему усмотрению, но нет доступа к базовому оборудованию в материальном виде и возможности заглянуть в него.
- *DBaaS (Database as a Service — база данных как услуга).* Сама MySQL является облачным ресурсом. Вы получаете учетные данные доступа к серверу MySQL. Вы можете конфигурировать некоторые параметры MySQL, но у вас нет доступа к базовой операционной системе или возможности заглянуть в нее или экземпляр виртуального сервера. Например, Amazon RDS работает с MySQL. Некоторые из этих служб на самом деле не содержат MySQL, но совместимы с протоколом MySQL и языком запросов.

В основном мы будем говорить о первой категории — облачном хостинге на таких платформах, как AWS, Rack-space Cloud и Joyent[1]. С одной стороны, есть много хороших ресурсов, обучающих развертыванию MySQL и управлению ею, а также ресурсов, необходимых для ее запуска, а с другой — платформ слишком много, чтобы

[1] Хорошо, хорошо, мы признаем это. Amazon Web Services — это облако. Эта глава в основном касается AWS.

мы могли их все охватить в этой книге. Поэтому не станем приводить примеры кода и обсуждать операционные методы. Вместо этого в данной главе уделим основное внимание ключевым различиям между запуском MySQL в облаке и традиционным развертыванием сервера, а также полученным экономическим и эксплуатационным характеристикам. Мы предполагаем, что вы знакомы с облачными вычислениями. Это не введение в данную тему — мы хотим помочь вам избежать некоторых подводных камней, с которыми вы можете столкнуться, если не являетесь экспертом по работе с MySQL в облаке.

В целом MySQL отлично работает в облаке. Запуск MySQL в облаке не отличается принципиально от запуска MySQL на любой другой платформе, однако у него есть несколько очень важных особенностей. Чтобы получить хорошие результаты, следует знать о них и соответствующим образом разрабатывать свое приложение и архитектуру. В некоторых случаях хостинг MySQL в облаке не очень подходящее решение, а иногда это лучшее изобретение человечества после хлеба, нарезанного для тостера. Но в большинстве случаев это просто еще одна платформа развертывания.

Важно понимать, что облако — это платформа развертывания, а не архитектура. На вашу архитектуру влияет платформа, но платформа и архитектура не одно и то же. Если вы путаете эти два понятия, то, скорее всего, сделаете плохой выбор, который может вызвать проблемы в будущем. Именно поэтому мы потратим массу времени на обсуждение важных отличий MySQL в облаке от традиционной модели.

Достоинства, недостатки и мифы облака

У облачных вычислений есть множество достоинств, часть из которых проявляется только тогда, когда они используются вместе с MySQL. На эту тему написано множество книг[1], поэтому не станем тратить на нее слишком много времени. Однако перечислим несколько важных моментов, поскольку вскоре собираемся обсудить недостатки облаков, но при этом не хотели бы, чтобы вы считали, что мы настроены по отношению к ним чрезмерно критично.

- Облако — это способ вынести часть инфраструктуры на аутсорсинг, так что вам не нужно будет управлять ею. Не придется приобретать оборудование и развивать связи внутри цепочек поставок, заменять неисправные жесткие диски и т. д.
- Облако, как правило, рассчитано на оплату по мере использования. Тем самым платежи, производимые в момент осуществления инвестиций, трансформируются в текущие операционные расходы.
- Облако постепенно становится дороже, поскольку поставщики услуг предоставляют новые сервисы и снижают затраты. Вам не нужно ничего делать самостоятельно (например, обновлять свои серверы), чтобы воспользоваться этими улучшениями, просто с течением времени появится больше вариантов (которые к тому же лучше предыдущих) по более низкой цене.

[1] См. книгу Джорджа Риза (George Reese) Cloud Application Architectures (издательство O'Reilly).

- Облако позволяет легко вводить в эксплуатацию серверы и другие ресурсы и выводить их по мере надобности, и вам не нужно будет что-то делать с ними по окончании эксплуатации или возмещать расходы, перепродавая их.
- Облако — это другой способ предоставления инфраструктуры как ресурсов, которые определяются и контролируются через API, и это позволяет гораздо лучше их автоматизировать. Эти преимущества можно получить и в частном облаке.

Конечно, не все в облаке так радужно. Перечислим некоторые недостатки, способные вызвать проблемы (чуть позже в этой главе отметим минусы, специфичные для MySQL).

- Ресурсы являются разделяемыми и непредсказуемыми, и вы можете получить даже больше, чем заплатите. Это приятный момент, но в то же время он может затруднить планирование мощности. Если вы получаете больше своей доли вычислительных ресурсов и не осознаете этого, существует риск, что кто-то другой будет требовать для себя справедливую долю ресурсов, уменьшая вашу производительность до такой, какой она должна была быть. В общем, довольно трудно точно понять, что именно вы должны получить, и при этом большинство провайдеров облачных хостингов не могут дать конкретных ответов на такие вопросы.
- Нет гарантии обеспечения мощности или доступности. Вы можете планировать ввод в эксплуатацию новых экземпляров, но что станете делать, если у провайдера переполнится лимит подписки? Так происходит во многих разделяемых ресурсах, может случиться и в облаке.
- В случае неполадок с виртуализованными и разделяемыми ресурсами устранить их может оказаться сложно, особенно потому, что у вас нет доступа к реальному оборудованию для проверки и измерения того, что происходит. Например, мы видели системы, в которых утилита iostat утверждала, что ввод/вывод в полном порядке или утилита vmstat показывала, что процессор работает как надо. А результат измерения времени, прошедшего до завершения выполнения задач, свидетельствовал, что ресурсы явно были перегружены чем-то еще в системе. Столкнувшись с проблемами производительности на облачной платформе, провести тщательные измерения даже более важно, чем обычно. Если вы плохо разбираетесь в этом, возможно, вам не удастся определить, это базовая система тормозит или вы сделали что-то, что заставляет приложение предъявлять необоснованные требования к ней.

Мы можем подытожить сказанное, заявив, что облако не слишком прозрачно и в нем хуже контроль над производительностью, доступностью и мощностью. Наконец, приведем несколько мифов об облаках, которые нужно учитывать.

- *Облако по своей природе более масштабируемо.* Приложения, их архитектуры и организации, которые ими управляют, масштабируемы или нет. Облако по своей природе не является масштабируемым только потому, что это облако, а выбор масштабируемой платформы автоматически не делает ваше приложение масштабируемым. Действительно, если у провайдера облачного хостинга не переполняется лимит подписки, то имеются ресурсы, которые вы можете приобрести, однако доступность ресурсов, когда они вам нужны, — это лишь один из аспектов масштабируемости.

- *Облако автоматически улучшает или даже гарантирует время безотказной работы.* На самом деле отдельные серверы с облачным хранилищем с большей вероятностью выйдут из строя или столкнутся с отключением электричества, чем хорошо продуманная выделенная инфраструктура. Однако многие этого не осознают. Например, один человек написал: «Мы модернизируем свою инфраструктуру до облачной системы, чтобы обеспечить 100 % времени безотказной работы и масштабируемость». Это случилось сразу после того, как в AWS произошли два огромных отключения электричества, которые повредили большие фрагменты базы ее пользователей. Хороший разработчик архитектуры может проектировать надежные системы с ненадежными компонентами, но в целом более надежная инфраструктура способствует повышению доступности. (Но, конечно, не 100 % времени безотказной работы.)

  В то же время, подписываясь на службу облачных вычислений, вы покупаете платформу, созданную экспертами. Они за вас позаботились о большом количестве низкоуровневых элементов, а это означает, что вы можете сосредоточиться на задачах более высокого уровня. Если вы строите собственную платформу и не являетесь экспертом по всем этим мелочам, то, вероятно, допустите ошибки новичка, из-за которых рано или поздно случится простой. Таким образом, облачные вычисления могут помочь увеличить время безотказной работы.

- *Облако — это единственное, что обеспечивает (вставьте здесь, что именно).* Многие преимущества облака унаследованы от технологий, используемых для создания облачных платформ, и могут быть получены и с облаком, и без него[1]. Например, при надлежащем управлении виртуализацией и планированием мощности вы можете развернуть новую машину так же легко и быстро, как и в любом облаке. Для этого вам не нужно облако.

- *Облако — это панацея.* Может показаться абсурдным, что кто-то действительно может это сказать, но некоторые говорят. Это ерунда.

Разумеется, облачные вычисления обеспечивают уникальные преимущества, и со временем мы лучше разберемся, что они собой представляют и когда полезны. Одно можно сказать наверняка: все это новинки, и любые прогнозы, сделанные сейчас, вряд ли мирно состарятся. Мы будем осторожны в своей книге и оставим прочие вопросы этой темы для личных обсуждений.

Экономика MySQL в облаке

В некоторых случаях облачный хостинг, безусловно, может быть экономичней, чем традиционное развертывание сервера. По нашему опыту, облачный хостинг — отличное решение для многих компаний, работающих на прототипах, или компаний, которые постоянно разворачивают новые концепции и в основном управляют ими через пробную версию. На ум сразу приходят разработчики мобильных приложе-

[1] Мы не утверждаем, что это будет легко или дешево. Мы просто говорим, что облако не единственная технология, с помощью которой можно получить эти преимущества.

ний и игр. Рынок этих технологий расширяется с распространением мобильных вычислений, и это быстро развивающийся мир. Во многих случаях факторы успеха не контролируются разработчиками. Это такие факторы, как устные рекомендации или время выхода, совпадающее с важными мировыми событиями.

Мы помогали многим компаниям создавать в облаке мобильные приложения, социальные сети и игровые программы. Одной из стратегий, которую многие из них используют, является разработка и выпуск приложений как можно быстрее и дешевле. Если приложение окажется удачным, компания будет инвестировать средства в его расширение, в противном случае работу над ним быстро прекратят. Некоторые компании создают и выпускают приложения с жизненным циклом всего лишь в несколько недель. В такой среде облачный хостинг не требует больших усилий.

Если у вас небольшая компания, вы, скорее всего, не можете позволить себе иметь собственный центр обработки данных с таким количеством оборудования, чтобы соответствовать кривой масштабирования популярного вирусного приложения для Facebook. Мы также помогали масштабировать некоторые из крупнейших приложений для Facebook. Удивительно, как быстро они растут — иногда кажется, что даже быстрее, чем некоторые хостинговые компании устанавливают серверы. Но что еще хуже, рост этих приложений совершенно непредсказуем: точно так же ими может заинтересоваться всего лишь парочка пользователей. Мы работали над такими приложениями как в центрах обработки данных, так и в облаке. Если у вас небольшая компания, облако может помочь уменьшить риск того, что придется масштабироваться больше и быстрее, чем позволяет капитал, которым вы располагаете.

Другим потенциально замечательным способом использования облака является запуск некритической инфраструктуры, такой как среда интеграции или тестовые стенды для развертывания и анализа. Предположим, что жизненный цикл вашего развертывания длится две недели. Испытываете ли вы его каждый час на протяжении всего этого времени или тестируете, приближаясь к финишу? Во многих случаях пользователям необходимо тестировать окружающую среду на стадиях разработки и обкатки только время от времени. В таких случаях облачный хостинг может помочь сэкономить деньги.

Расскажем о двух способах, какими мы сами используем облака. Первый из них является частью процесса интервью при приеме на работу технического персонала, в ходе которого мы просили претендентов решить какие-то реальные проблемы. Мы развертывали «сломанные» машины с Amazon Machine Images (AMI), который создали для этой цели, и просили их войти в систему и выполнить разнообразные задачи на серверах. Мы не должны открывать доступ к нашей собственной сети, а этот прием необыкновенно дешев. Другой способ задействования облачного хостинга состоит в использовании серверов разработки и обкатки для новых проектов. Один проект запускается на сервере обкатки в облаке в течение нескольких месяцев. Общий счет за эту услугу составляет менее одного доллара! Мы никак не можем добиться того же на собственной инфраструктуре. Простая отправка электронной почты нашему системному администратору с просьбой о создании сервера для обкатки потребует времени, стоимость которого превышает один доллар.

В то же время облачный хостинг может оказаться более дорогим в долгосрочной перспективе. Вы должны потратить время и выполнить самостоятельные вычисления, если планируете долго его использовать. Это потребует провести эталонное тестирование и учесть общую стоимость владения им (total cost of ownership, TCO), а также попытаться спрогнозировать, какой будет отдача от инноваций как в облачные вычисления, так и в стандартное оборудование. Чтобы принять правильное решение, учитывая все важные детали, вам нужно свести все к одному числу — количеству бизнес-транзакций, выполняемых за доллар. Все меняется быстро, поэтому сами расчеты мы оставляем в качестве упражнения для читателей.

Масштабирование MySQL и HA в облаке

Как мы уже отмечали, MySQL автоматически не становится более масштабируемой в облаке. Фактически менее мощные доступные машины намного раньше заставляют использовать стратегии горизонтального масштабирования. В свою очередь, облачные серверы менее надежны и предсказуемы, чем выделенное оборудование, поэтому для достижения высокой доступности требуется изобретательность.

По большому счету, не так уж много различий между масштабированием MySQL в облаке и обычным ее масштабированием. Самое большое состоит в возможности получения серверов по требованию. Однако существует ряд ограничений, которые затрудняют масштабирование и обеспечение высокой доступности, по крайней мере в некоторых облачных средах. Например, в облачной платформе AWS вы не можете использовать эквивалент виртуальных IP-адресов для быстрого выполнения аварийного переключения. Ограниченный контроль над этими ресурсами просто означает, что вам нужно использовать другие подходы, такие как прокси. (Стоит обратить внимание на ScaleBase.)

Другая замануха — это мечта об автоматическом масштабировании, то есть развертывании большего количества экземпляров при возрастании спроса и их отключении, когда спрос уменьшится. Этого можно добиться с помощью частей стека без сохранения состояния, таких как веб-серверы, однако очень сложно сделать с сервером базы данных, потому что он сохраняет состояние. В особых случаях, таких как использование приложений, предназначенных для чтения, вы можете получить упрощенную форму автоматического масштабирования с помощью добавления подчиненных серверов[1], но это решение не на все случаи жизни. На практике, несмотря на то что многие приложения могут использовать автомасштабирование на веб-уровне, MySQL не способна работать в кластере серверов без общего хранилища, в котором все серверы одноранговые.

Это можно сделать с помощью архитектуры шардов, которая автоматически шардируется заново и растет или сжимается[2] по мере необходимости, однако MySQL сама по себе автоматически масштабироваться не может.

---

[1] Популярный сервис с открытым исходным кодом для автоматического масштабирования репликации MySQL в облаке Scalr (http://scalr.net).

[2] Это то, что специалисты по информатике любят называть нетривиальным вызовом.

Фактически, поскольку сервер MySQL обычно основной или единственный компонент приложения с сохранением состояния, для людей вполне типично перемещать приложение в облако из-за преимуществ, которые оно предоставляет всем компонентам, кроме базы данных, — веб-серверам, серверам очереди заданий, кэшу и т. п. И MySQL просто вынуждена перемещаться туда, куда идут остальные компоненты.

В конце концов база данных не является центром мироздания. Если преимущества для остальных компонентов приложения перевешивают дополнительные затраты и усилия, необходимые для того, чтобы заставить MySQL работать так, как надо, тогда вопрос не в том, произойдет ли это в принципе. Вопрос будет звучать так: «Как это будет происходить?» Чтобы ответить на него, следует понять, с какими еще проблемами вы можете столкнуться в облаке. Обычно они сосредоточены вокруг ресурсов, доступных серверу базы данных.

Четыре основных ресурса

Для выполнения своей работы MySQL нуждается в четырех основных ресурсах: циклах процессоров, памяти, вводе/выводе и сети. В большинстве облачных платформ эти ресурсы имеют характерные и важные различия. Одним из полезных способов принятия решения о размещении MySQL в облаке является изучение этих различий и их последствий для MySQL.

- Процессоры, как правило, меньше и медленнее. Крупнейшие стандартные экземпляры EC2 на момент написания книги предлагают восемь виртуальных ядер процессора. Виртуальные процессоры EC2 работают значительно медленнее, чем лучшие процессоры (некоторые тонкости будут рассмотрены в этой главе чуть позже вместе с нашими эталонными тестами). Вероятно, это довольно типично для большинства облачных хостингов, хотя могут быть вариации. EC2 предлагает экземпляры с большим количеством ресурсов процессора, но у них меньше максимальный объем памяти. На момент написания книги в стандартных серверах используются десятки ядер процессора — и даже больше, если учитывать аппаратные потоки[1].
- Объем памяти ограничен. Самые большие экземпляры EC2 в настоящее время предлагают 68,4 Гбайт памяти. В свою очередь, стандартные серверы предлагают от 512 Гбайт до 1 Тбайт памяти.
- Производительность ввода/вывода ограничена пропускной способностью, задержкой и согласованностью. Существует два варианта хранения в облаке AWS.

  Первый использует тома EBS, которые похожи на облачную SAN. Хорошим приемом в облаке AWS является создание серверов в томе RAID 10 поверх EBS. Однако EBS — это разделяемый ресурс, как и сетевое соединение между серверами EC2 и EBS. Задержка может быть высокой и непредсказуемой даже при умеренной пропускной способности. Задержка ввода/вывода для устройств EBS, согласно

[1] Стандартное оборудование по-прежнему предлагает больше возможностей, чем MySQL, оно может эффективно использоваться с точки зрения ЦП, ОЗУ и ввода/вывода, поэтому сравнивать облако с самой большой мощностью, доступной вне облака, нецелесообразно.

нашим измерениям, оказалась равной десятым долям секунды. Для сравнения: напрямую подключенные стандартные жесткие диски продемонстрировали реакцию в несколько миллисекунд, а флеш-устройства были на порядок быстрее жестких дисков. Однако у томов EBS есть множество приятных функций, таких как интеграция с другими службами AWS, быстрая фиксация состояния и т. п.

Второй вариант хранения — локальное хранилище экземпляра. У каждого сервера EC2 есть локальное хранилище некоторого объема, которое фактически привязано к базовому серверу. Оно может обеспечить более стабильную производительность, чем EBS[1], но при остановке экземпляра данные теряются. Эфемерный характер локального хранилища делает его непригодным в большинстве случаев.

- Производительность сети обычно вполне приличная, однако она является разделяемым ресурсом и может варьироваться. Вы можете добиться более быстрой работы и стабильной производительности сети при использовании стандартного оборудования, однако процессор, ОЗУ и ввод/вывод часто оказываются узкими местами. При этом у нас никогда не было проблем с производительностью сети в облаке AWS.

Как можно заметить, три из четырех основных ресурсов ограничены в облаке AWS, и в некоторых случаях весьма значительно. В целом основные ресурсы в облаке не так мощны, как то, что доступно на стандартном оборудовании. Конкретные последствия этого обсудим в следующем разделе.

Производительность MySQL в облачном хостинге

В целом производительность MySQL на облачных платформах, таких как AWS, меньше той, которой можно добиться в другом месте, из-за более слабой производительности процессора, памяти и ввода/вывода. Конкретные показатели варьируются от одной облачной платформы к другой, но общая тенденция не меняется[2]. Тем не менее облачный хостинг может оказаться платформой, по-прежнему предоставляющей уровень производительности, достаточный для ваших потребностей. Естественно, он лучше подходит для одних нужд, чем для других.

Неудивительно, что, используя в облаке более слабые ресурсы для базы данных, вы не можете заставить MySQL работать так же быстро, как и вне его. Но, возможно, вас удивит тот факт, что вы не сможете заставить ее работать так же быстро, как на физическом оборудовании с аналогичными параметрами. Например, если вы ис-

---

[1] На самом деле для экземпляра не выделяется локальное хранилище до момента записи в него, что приводит к издержкам при первой операции записи для каждого записываемого блока данных. Чтобы избежать этих издержек, можно предварительно заполнить все устройство данными с помощью утилиты dd.

[2] Если вы верите http://www.xkcd.com/908/, то очевидно, что у всех облаков имеются одни и те же недостатки. Мы просто озвучили.

пользуете сервер с восемью ядрами процессора, 16 Гбайт памяти и RAID-массивом среднего уровня, то можете предположить, что аналогичной производительности можно добиться от экземпляра EC2 с восемью вычислительными единицами EC2, 15 Гбайт памяти и несколькими томами EBS. Однако аналогичная производительность отнюдь не гарантирована. Производительность экземпляра EC2, по всей видимости, будет варьироваться сильнее, чем производительность физического оборудования, особенно потому что этот экземпляр не является супербольшим и, по-видимому, на одном физическом оборудовании разделяет ресурсы с другими экземплярами.

Переменная производительность — это действительно большая проблема. MySQL и InnoDB, в частности, ее не любят, особенно переменную производительность ввода/вывода. Операции ввода/вывода могут захватывать блокировки мьютекса внутри сервера, и если так происходит в течение длительного времени, могут начаться сбои, которые проявляются в большом количестве застрявших процессов, необъяснимо длительных запросах и всплесках значений таких переменных состояния, как `Threads_running` или `Threads_connected`.

Практический результат изменчивой или непредсказуемой производительности заключается в том, что очередь становится более серьезной. Она является естественным следствием изменчивости времени отклика и длительности промежутков между доступом к ресурсу. Существует раздел математики, посвященный теории очередей. Все компьютеры представляют собой сети очередей, а запросы на ресурсы должны ждать, если ресурс (ЦП, ввод/вывод, сеть и т. п.) занят. Когда производительность ресурсов более изменчива, запросы чаще пересекаются и стоят в большем количестве очередей. В результате на большинстве платформ облачных вычислений немного сложнее добиться высокой конкурентности или постоянного низкого времени отклика. У нас есть большой опыт наблюдения за этими ограничениями на платформе EC2. По нашему опыту, наибольшая конкурентность, которой можно ожидать от MySQL при максимальных размерах экземпляра, — это величина `Threads_running` от 8 до 12 при типичных рабочих нагрузках сети OLTP. Как правило, при более высокой конкурентности производительность становится неприемлемой.

Обратите внимание, что мы сказали «типичные рабочие нагрузки сети OLTP». Не все рабочие нагрузки одинаково реагируют на ограничения облачных платформ. Оказывается, на самом деле есть рабочие нагрузки, отлично работающие в облаке, в то же время в некоторых случаях производительность падает особенно сильно. Давайте посмотрим, о чем идет речь.

- Как мы только что обсудили, рабочие нагрузки, требующие высокой конкурентности, не слишком хорошо подходят для облачных вычислений. То же самое относится к приложениям, требующим чрезвычайно быстрого времени отклика. Причина сводится к ограниченным числу и скорости виртуальных процессоров. Каждый запрос выполняется на одном процессоре внутри MySQL, поэтому время отклика запроса ограничено чистой скоростью процессора. Если требуется короткое время отклика, то нужны быстрые процессоры. Чтобы поддерживать высокий уровень конкурентности, нужно много процессоров. Это правда, что MySQL и InnoDB не обеспечивают большую отдачу на вложенный доллар

на многих десятках процессорных ядер, но в целом в настоящее время они, как правило, хорошо горизонтально масштабируются, по крайней мере до 24 ядер. Как правило, это больше, чем вы можете получить в облаке.

- При рабочей нагрузке, характеризующейся большим объемом ввода/вывода, производительность в облаке снижается. Если операции ввода/вывода медленные и их объем меняется, работа быстро останавливается. В то же время, если рабочая нагрузка не требует множества операций ввода/вывода по пропускной способности (количеству операций в секунду) или по ширине полосы (количеству байт в секунду), MySQL может работать как часы.

Предыдущие пункты фактически вытекают из слабых сторон процессора и ресурсов ввода/вывода в облаке. Что вы можете с этим поделать? С ограничениями процессора сделать можно немного. Если ресурса процессора не хватает, значит, его не хватает. Однако система ввода/вывода — это совсем другое дело. Фактически ввод/вывод — это обмен между двумя типами памяти: энергозависимой (ОЗУ) и постоянной (диск, EBS или что там у вас есть). В результате на запросы ввода/вывода MySQL может влиять объем памяти системы. Если его достаточно, чтение может производиться из кэша, тем самым количество операций ввода/вывода, необходимых для чтения и записи, уменьшается. Обычно операции записи могут быть записаны в буфер памяти, а несколько записей в одни и те же биты памяти могут быть объединены, а затем выполнены с помощью одной операции ввода/вывода.

Вот здесь на первый план выходят ограничения на память. При объеме памяти, достаточном для хранения рабочего множества данных[1], необходимость в операциях ввода/вывода для определенных рабочих нагрузок может значительно сократиться. Более крупные экземпляры EC2 также обеспечивают лучшую производительность сети, что способствует улучшению операций ввода/вывода в тома EBS. Но если ваше рабочее множество слишком велико и не помещается в самые большие доступные экземпляры, потребность в операциях ввода/вывода возрастает в геометрической прогрессии и все начинает блокироваться и стопориться так же, как описывалось ранее. Самые большие экземпляры со значительным объемом памяти в EC2 имеют достаточно памяти, чтобы хорошо обслуживать серьезную рабочую нагрузку. Однако вы должны помнить, что «прогрев» может продолжаться очень долго (подробнее об этой теме позже в этом разделе).

Какие типы рабочей нагрузки нельзя улучшить, добавив больше памяти? Независимо от буферизации некоторые рабочие нагрузки со значительным объемом операций записи просто нуждаются в большем количестве операций ввода/вывода, чем можно ожидать от большинства платформ облачных вычислений. Например, если выполняется множество транзакций в секунду, которые для надежности требуют много операций ввода/вывода, вы можете добиться только такой пропускной способности системы, которую может дать EBS. Аналогично если вы загружаете большое количество данных в базу, то можете превысить доступную ширину полосы.

---

[1] См. главу 9 для определения рабочего множества и обсуждения того, как оно влияет на операции ввода/вывода.

Возможно, вы считаете, что можно улучшить производительность ввода/вывода, разделив и зеркалируя тома EBS с помощью RAID. Это поможет, но только до определенной степени. Проблема состоит в том, что, добавляя тома EBS, вы фактически увеличиваете вероятность того, что в какой-то момент один из них станет плохо работать. В свою очередь, способ работы операций ввода/вывода внутри InnoDB приводит к тому, что слабое звено становится узким местом всей системы. Мы опробовали десять наборов RAID 10 и 20 томов EBS, и RAID с 20 томами имел больше проблем со стопором, чем десятитомный. Измерив производительность ввода/вывода базовых блочных устройств, мы выяснили, что только один или два тома EBS работали медленно, тем не менее пострадал весь сервер.

Чтобы уменьшить потребность в операциях ввода/вывода, можно внести изменения в приложение и сервер. Тщательное логическое и физическое проектирование базы данных (схема и индексирование) сильно влияет на сокращение количества операций ввода/вывода, а также оптимизацию приложений и запросов. Это самые могучие рычаги из тех, что можно использовать для сокращения количества операций ввода/вывода. При некоторых рабочих нагрузках, например характеризующихся большим количеством вставок, может помочь разумное использование секционирования для концентрации операций ввода/вывода в одной секции, индексы которой помещаются в память. Вы можете уменьшить долговечность, например установив `innodb_flush_logs_at_trx_commit = 2` и `sync_binlog = 0` или перемещая журналы транзакций InnoDB и двоичные журналы из томов EBS на локальные диски (хотя это и опасно). Но чем больше вы попытаетесь выдоить из сервера, тем больше сложностей (и, следовательно, затрат) неизбежно добавите.

Кроме того, можно обновить программное обеспечение сервера MySQL. Последние версии MySQL и InnoDB (последние версии MySQL 5.1 с плагином InnoDB или MySQL 5.5 и новее) предлагают значительно улучшенную производительность операций ввода/вывода и меньшее количество внутренних узких мест. При этом они будут реже испытывать проблемы со стопором и нагромождениями, чем старый код в ранней версии 5.1 и предыдущих. Percona Server может предложить еще больше преимуществ при определенных нагрузках. Например, возможность Percona Server быстро прогреть буферный пул после перезагрузки чрезвычайно помогает в быстром восстановлении работы сервера, особенно если производительность операций ввода/вывода невелика, а сервер полагается на работу в памяти. Это один из рассмотренных нами сценариев обеспечения хорошей производительности в облаке, поскольку облачные серверы чаще выходят из строя, чем физическое оборудование. Percona Server может сократить время «прогрева» от часов или даже дней до нескольких минут. На момент написания книги аналогичные возможности были доступны в разрабатываемом релизе MySQL 5.6.

Однако в конечном счете растущее приложение достигнет точки, когда вам придется шардировать базу данных, чтобы остаться в облаке. Мы стараемся максимально избегать шардирования, но, когда у вас ограничены лошадиные силы, в какой-то момент придется либо уйти в другое место (покинуть облако), либо разбить целое на много маленьких кусочков, требования которых не превышают возможностей

доступного виртуального оборудования. Обычно предполагается, что необходимость шардирования возникает тогда, когда рабочее множество перестает помещаться в память, что означает 50–60 Гбайт на крупнейших экземплярах EC2. Имея большой опыт работы с многотерабайтными базами данных на физическом оборудовании, мы можем утверждать: при работе в облаке необходимость шардирования наступает гораздо раньше.

Эталонное тестирование MySQL в облаке. Мы провели несколько эталонных тестов, чтобы проиллюстрировать производительность MySQL в облачной среде AWS. Если требуется выполнять много операций ввода/вывода, то практически невозможно получить согласованные и воспроизводимые эталонные тесты в облаке, поэтому мы выбрали работу в памяти и измерили практически все, кроме операций ввода/вывода. Мы использовали Percona Server версии 5.5.16 с четырехгигабайтным буферным пулом для запуска стандартного эталонного теста SysBench в режиме «только для чтения» на 10 миллионов строк данных. Это позволило сравнить результаты при различных размерах экземпляров. Мы исключили экземпляры с мощными процессорами, поскольку они фактически имеют меньшую мощность процессора, чем экземпляр m2.4xlarge, а в качестве ориентира задействовали сервер Cisco. Машина Cisco была довольно мощная, но немного устаревшая, с двумя процессорами Intel Xeon X5670 Nehalem с тактовой частотой 2,93 ГГц. Каждый процессор содержал шесть ядер с двумя аппаратными потоками на каждом, которые операционная система видела в общей сложности как 24 процессора. На рис. 13.1 показаны результаты.

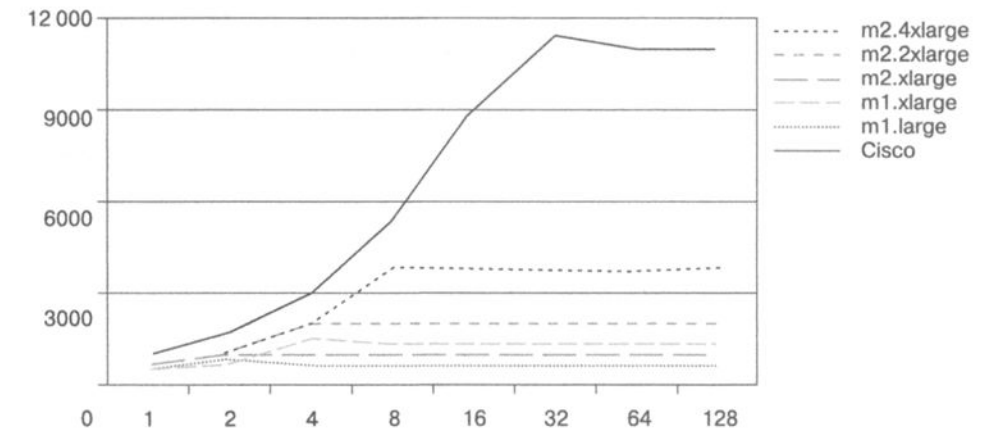


Рис. 13.1. Эталонное тестирование MySQL в облаке AWS при помощи инструмента SysBench в режиме «только для чтения»

Если учитывать рабочую нагрузку и аппаратное обеспечение, результаты удивлять не должны. Например, самый большой экземпляр EC2 достигает пика на восьми потоках, поскольку он имеет восемь ядер процессора. (При связанной с чтением/записью нагрузке часть времени экземпляр выполняет ввод/вывод, не задействуя CPU, так что

мы фактически получим более восьми конкурентных потоков.) На основе этой диаграммы можно прийти к выводу, что преимущество Cisco заключается в мощности процессора, как мы и думали. Чтобы в этом убедиться, мы провели эталонное тестирование производительности только процессора, используя эталонный тест prime-number утилиты SysBench. На рис. 13.2 представлены результаты.

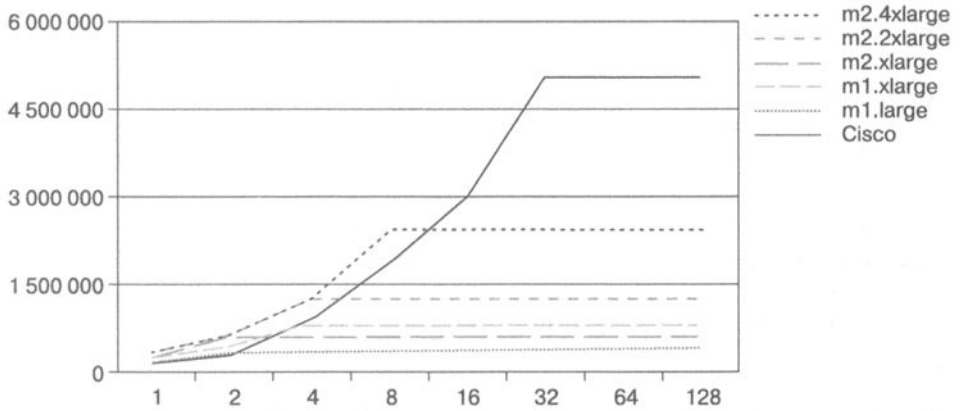


Рис. 13.2. Эталонное тестирование тестом prime-number процессора для серверов AWS при помощи инструмента SysBench

Производительность в расчете на процессор у сервера Cisco меньше, чем у серверов EC2. Удивлены? Мы были немного удивлены. Эталонный тест prime-number — это, по сути, исходные инструкции процессора, и поэтому он не должен иметь заметных накладных затрат на виртуализацию или большой объем трафика. Таким образом, объяснить полученные результаты, вероятно, можно так: процессорам сервера Cisco уже пара лет, и они медленнее, чем процессоры серверов EC2. Но при выполнении более сложных задач, таких как запуск сервера базы данных, накладные затраты на виртуализацию ухудшают производительность серверов EC2. Не всегда легко разграничивать медленные процессоры, медленный доступ к памяти и накладные затраты на виртуализацию, но в данном случае различия кажутся довольно очевидными.

База данных MySQL как услуга

Установка MySQL на облачном сервере не является единственным вариантом ее использования в облаке. Все больше компаний предлагают саму базу данных в качестве облачного ресурса, называя ее «база данных как услуга» (DBaaS, а иногда и DaaS). Это означает, что вы можете использовать базу данных где-нибудь в облаке, а ее фактическое обслуживание оставить кому-то другому. Хотя в этой главе мы посвятили большую часть времени изучению IaaS, рынок IaaS быстро становится массовым и мы ожидаем, что в будущем упор будет сделан на DBaaS. На момент написания книги существовало несколько поставщиков DBaaS.

Amazon RDS

Мы видели гораздо больше развертываний в сервисе реляционных баз данных Amazon (Relational Database Service, RDS), чем в любых других сервисах DBaaS. Amazon RDS — это не только совместимый с MySQL сервис, это на самом деле MySQL, поэтому он полностью совместим с вашим собственным сервером MySQL[1] и может служить его упрощенной заменой. Мы не знаем наверняка, но, как и большинство людей, полагаем, что RDS размещается на машине EC2, поддерживаемой томами EBS. Amazon официально не подтвердила базовые технологии, но, поскольку мы хорошо знаем RDS, нам кажется довольно очевидным, что это просто MySQL, EC2 и EBS.

Amazon выполняет за вас все системное администрирование. Вы не имеете доступа к машине EC2 — только учетные данные для входа в MySQL и все. Вы можете создавать базы данных, вставлять данные и т. д. Вы не заблокированы, если хотите, можете экспортировать данные и перемещать их в другое место, а также создавать копии томов и монтировать их на других машинах.

В RDS есть ряд ограничений, не позволяющих вам изучать то, как Amazon управляет сервером или хостом, или вмешиваться в процесс. Например, существуют некоторые ограничения привилегий. Вы не можете использовать `SELECT INTO OUTFILE`, `FILE()`, `LOAD DATA INFILE` или любой другой способ доступа к файловой системе сервера через MySQL. Не можете сделать ничего связанного с репликацией и не можете расширить свои привилегии, чтобы предоставить себе эти права. Для предотвращения этого Amazon включила триггеры в системные таблицы. И принимая условия обслуживания, вы соглашаетесь не пытаться обойти эти ограничения.

Установленную версию MySQL слегка изменили, чтобы не дать вам вмешаться в работу сервера, но в остальном это, судя по всему, всем знакомая MySQL. Мы провели эталонное тестирование RDS с EBS и EC2 и не обнаружили никаких отличий, кроме зависящих от платформы. То есть, похоже, Amazon не предприняла никаких улучшений производительности сервера.

В зависимости от конкретных условий RDS может предложить несколько неоспоримых преимуществ.

- Amazon выполняет за вас работу системного администратора и даже большую часть администрирования базы данных. Например, обрабатывает репликацию и гарантирует, что вы ее не испортите.
- В зависимости от структуры затрат и кадровых ресурсов RDS может быть дешевле альтернативных вариантов.
- Ограничения можно рассматривать как плюсы: Amazon забирает у вас заряженный пистолет, из которого вы могли бы выстрелить себе в ногу.

---

[1] Если вы не используете альтернативную подсистему хранения или какую-либо другую нестандартную модификацию для MySQL.

Однако у нее есть несколько потенциальных недостатков.

- Поскольку вы не можете получить доступ к серверу, то не знаете, что происходит в операционной системе. Например, не можете измерить время отклика операций ввода/вывода или уровень загрузки процессора. Amazon обеспечивает эти возможности с помощью другого своего сервиса, CloudWatch. Он предоставляет достаточно подробные характеристики, пригодные для поиска и устранения многих проблем с производительностью, но иногда вам нужны исходные данные, чтобы точно знать, что происходит. (Также вы не можете использовать такие функции, как `FILE()`, для доступа к `/proc/diskstats`.)
- Нельзя получить полный файл журнала медленных запросов. Вы можете указать MySQL журналировать медленные запросы в таблицу журналирования CSV, но это хуже. Она потребляет гораздо больше ресурсов сервера и предоставляет менее точную информацию о времени отклика. Это затрудняет профилирование и устранение неполадок SQL.
- Если вам нужны самые последние и самые грандиозные усовершенствования или некоторые улучшения производительности, например те, что можно получить с Percona Server, вам не повезло. RDS их не предлагает.
- Вам приходится полагаться на службу поддержки Amazon для решения некоторых проблем, которые вы могли бы исправить самостоятельно. Например, предположим, что запросы зависают или ваш сервер выходит из строя из-за повреждения данных. Можете либо ждать, пока Amazon это исправит, либо взять дело в свои руки. Но чтобы сделать последнее, вы должны получить данные где-то в другом месте, так как не можете для исправления экземпляра получить к нему доступ. Вам придется потратить дополнительное время и заплатить за дополнительные ресурсы, если хотите что-то сделать. Это не просто теоретизирование — нам поступало множество просьб о помощи в ситуациях, когда для устранения неполадок действительно нужен доступ к системе. Но в результате пользователи RDS не могли решить проблемы.

Что касается производительности, то, как мы уже говорили, RDS вполне сопоставима с большим экземпляром EC2 со значительным объемом памяти, хранилищем EBS и аналогичной MySQL. Вы можете выжать немного больше производительности из облака AWS, если напрямую используете EC2 и EBS, а также установите и настроите более высокопроизводительную версию MySQL, такую как Percona Server. Однако производительность вырастет не на порядок. Понимая это, имеет смысл основывать решение об использовании RDS для своих бизнес-потребностей не на производительности. Если она для вас действительно важна, не стоит вообще использовать облако AWS.

Другие решения DBaaS

Amazon RDS не единственный вариант DBaaS для пользователей MySQL. Существуют и такие сервисы, как FathomDB (http://fathomdb.com) и Xeround (http://xeround.com). Однако у нас недостаточно опыта, чтобы рекомендовать один из них, поскольку мы не выполняли в них никаких производственных развертываний.

Широкой публике доступна ограниченная информация о них. На ее основании можно сделать вывод, что FathomDB, по-видимому, похож на Amazon RDS, хотя доступен и в облаке Rackspace, и в облаке AWS. На момент написания книги он существовал в бета-версии для частного использования.

Xeround — совсем другое дело. Это распределенный кластер серверов с интерфейсом MySQL с запатентованной подсистемой хранения. Судя по всему, либо у него есть небольшая несовместимость с MySQL, либо он отличается от стандартной версии MySQL. Однако стабильная версия (GA) появилась сравнительно недавно, так что судить об этом продукте слишком рано. Подсистема хранения, по-видимому, взаимодействует с внутренней кластерной системой, которая имеет сходство с NDB Cluster. Кроме того, у него есть возможность автоматического шардирования, по мере увеличения и уменьшения рабочей нагрузки узлы добавляются и удаляются (динамическое масштабирование).

Существует множество других сервисов DBaaS, да и новые появляются довольно часто. Все, что мы пишем о них, устареет к тому моменту, когда вы это прочтете, поэтому мы оставляем вам возможность исследовать это поле самостоятельно.

Итоги главы

Существует как минимум два основных способа использования MySQL в облаке: установить ее на облачных серверах или применить возможности DBaaS. MySQL хорошо работает в облачном хостинге, но ограничения облачной среды обычно приводят к шардированию намного раньше, чем это становится необходимым за пределами облака. Кроме того, облачные серверы, которые кажутся сопоставимыми с существующим реальным оборудованием, скорее всего, приведут к снижению производительности и качества обслуживания.

Кажется, что иногда люди говорят: «Облако — это ответ. А в чем вопрос?» Это одна из крайностей, но тот, кто горячо верует, что облако — это панацея от всех трудностей, скорее всего, будет иметь проблемы. Три из четырех фундаментальных ресурсов, необходимых базе данных (процессор, память и диск), вероятно, будут значительно слабее и/или менее эффективны в облаке, что напрямую влияет на производительность MySQL.

Тем не менее MySQL отлично работает в облаке при большинстве рабочих нагрузок. В целом все будет в порядке, если вы сможете поместить свое рабочее множество в память и не создадите бóльшую рабочую нагрузку на запись, чем сможет обработать облачный ввод/вывод. Тщательно спроектировав архитектуру, а также выбрав подходящую версию MySQL и правильно ее сконфигурировав, вы сможете уравнять нагрузку и возможности вашей базы данных с сильными сторонами облака. Однако по своей природе MySQL не является облачной базой данных, то есть не может использовать все теоретически возможные преимущества облачных вычислений, например автомасштабирование. Альтернативные технологии, такие как Xeround, пытаются устранить эти недостатки.

Здесь так много говорилось о недостатках облака, что может создаться впечатление: мы против облаков. Это не так. Просто мы пытаемся сосредоточиться на MySQL, а не перечислять все преимущества облачных вычислений, так как это не сильно отличалось бы от того, что вы можете прочесть в других источниках. Мы попытались рассказать, чем отличается запуск MySQL в облаке и что вам действительно нужно о нем знать.

Наибольшие успехи, которые мы отмечали при облачных вычислениях, достигались тогда, когда облако использовали по коммерческим соображениям. Даже если стоимость одной бизнес-транзакции в долгосрочной перспективе будет выше в облаке, другие факторы, такие как повышенная гибкость, уменьшение первоначальных затрат, сокращение времени выхода на рынок и снижение риска, могут оказаться более важными. И преимущества для других элементов ваших приложений (не MySQL) могут компенсировать любые недостатки, связанные с использованием MySQL в облаке.

14 Оптимизация на уровне приложения

Если вы потратили много времени на улучшение производительности MySQL, ваш взгляд легко может замылиться и вы просто забудете обратить внимание на опыт пользователя. Можно вернуться на шаг назад и осознать: MySQL настолько оптимизирована, что вносит лишь небольшую лепту в общее время отклика. Таким образом, пришло время сфокусироваться на какой-то другой проблеме. Это отличная мысль, особенно для администратора базы данных, и именно то, что нужно сделать. Но что же вызывает проблемы, если не MySQL? Ответ можно найти наиболее надежным и быстрым способом, проведя измерения с использованием методов, показанных в главе 3. Если вы тщательно профилируете и следуете логике процессов, вам не составит труда найти корень проблемы. Однако иногда, даже если проблема связана с MySQL, проще всего решить ее в другой части системы!

Независимо от того, в чем заключается причина проблемы, обязательно найдется хоть один отличный инструмент, который сможет ее измерить, часто даже бесплатно. Например, если у вас проблемы с JavaScript или отображением страниц, можно использовать профилировщик, включенный в расширение Firebug для браузера Firefox, либо инструмент YSlow от Yahoo!. Мы называли несколько инструментов уровня приложения в главе 3. Некоторые инструменты даже профилируют весь стек; New Relic является примером инструмента, который профилирует интерфейс, приложение и серверную часть сетевых приложений.

Типичные проблемы

При анализе различных приложений мы постоянно натыкаемся на одни и те же проблемы. Часто причиной их возникновения становится использование при создании приложений плохо спроектированных готовых систем или популярных сред, упрощающих разработку. Иногда действительно проще и быстрее воспользоваться чем-то написанным другими людьми, однако на этом пути вас подстерегает опас-

ность: вы не знаете точно, как все устроено. Приведем перечень того, на что следует обращать внимание.

- *Кто именно потребляет ресурсы процессора, диска, сети и памяти на каждом из составляющих систему компьютеров?* Выглядят ли показатели, отражающие потребление, разумными? Если нет, взгляните пристальнее на приложения — основные пожиратели ресурсов. Часто простейшим способом решения проблемы является изменение конфигурации. Например, если сервер Apache израсходовал всю память, потому что создал 1000 рабочих процессов по 50 Мбайт каждый, то можно сконфигурировать приложение так, чтобы для него требовалось меньше рабочих процессов. Или сконфигурировать систему так, чтобы каждый рабочий процесс потреблял меньше памяти.
- *Действительно ли приложение использует все данные, которые получает?* Делать выборку 1000 строк только для того, чтобы вывести десять из них, а остальные отбросить, — очень распространенная ошибка (однако, если приложение кэширует остальные 990 строк, планируя использовать их в будущем, это может быть преднамеренной оптимизацией).
- *Выполняет ли приложение обработку, которую следовало бы сделать в базе данных, или наоборот?* Два примера: выборка всех строк из таблицы для того, чтобы подсчитать их количество, и выполнение сложных манипуляций со строками средствами базы данных. База данных прекрасно справляется с подсчетом строк, тогда как прикладным языкам программирования нет равных в работе с регулярными выражениями. Используйте подходящий инструмент.
- *Не слишком ли много запросов выполняет приложение?* Часто в возникновении такой ситуации следует винить интерфейсы на базе объектно-реляционных отображений (ORM), которые «избавляют программиста от необходимости писать SQL-код». Сервер базы данных изначально спроектирован для сопоставления данных из нескольких таблиц. Уберите вложенные циклы из кода и напишите вместо них запрос с соединением таблиц.
- *Не слишком ли мало запросов выполняет приложение?* Только что мы сказали, что слишком большое количество запросов — это зло. Но иногда ручное соединение и иные подобные приемы могут оказаться полезными, так как позволяют реализовать более детальное и эффективное кэширование, уменьшить количество блокировок, а то и ускорить работу приложения за счет эмуляции хеш-соединений в прикладном коде (применяемый в MySQL метод соединения на основе вложенных циклов не всегда эффективен).
- *Не подключается ли приложение к MySQL понапрасну?* Если можно получить данные из кэша, может, не стоит устанавливать соединение?
- *Не подключается ли приложение слишком часто к одному и тому же экземпляру MySQL, потому что разные его части открывают собственные соединения?* Обычно гораздо лучше использовать одно и то же соединение на протяжении всей работы.

- *Не выполняет ли приложение массу мусорных запросов?* Типичный пример — проверка состояния сервера непосредственно перед запросом или выбор нужной базы данных перед каждым из них. Возможно, было бы лучше всегда подключаться к одной базе, но для таблиц указывать полные имена (заодно упростится анализ запросов в журнале или с помощью команды `SHOW PROCESSLIST`, так как их можно будет выполнить, не изменяя текущую базу). Еще одна типичная проблема — подготовка соединения. В частности, драйвер в языке Java в это время выполняет массу операций, большую часть которых можно подавить. Другой распространенный источник мусорных запросов — команда `SET NAMES UTF8`, которая в любом случае не нужна (она не изменяет кодировку в клиентской библиотеке, а воздействует только на сервер). Если приложению для выполнения большей части работы нужна конкретная кодировка, то можно не устанавливать ее при каждом запросе, а назначить в качестве значения по умолчанию.
- *Использует ли приложение пул соединений?* Это может быть и хорошо и плохо. С одной стороны, это позволяет ограничить количество соединений, что неплохо, когда на одном соединении выполняется немного запросов (типичный пример — приложения на основе технологии Ajax). С другой — возможны побочные эффекты, например, из-за того, что приложение вмешивается в чужие транзакции, временные таблицы, изменяет чужие настройки соединения и переменные, определенные пользователем.
- *Применяются ли в приложении постоянные соединения?* Если да, то в результате этого может образовываться слишком много соединений с MySQL. В общем случае так делать не следует, за исключением ситуации, когда подключение к MySQL становится слишком затратным из-за медленной сети, или когда соединение используется всего для одного-двух быстрых запросов, или когда подключение производится настолько часто, что не хватает номеров локальных портов. Если MySQL сконфигурирована правильно, то необходимости в постоянных соединениях может и не возникнуть. Применяйте параметр `skip-name-resolve`, чтобы подавить обратные DNS-запросы, устанавливайте достаточно большое значение параметра `thread_cache` и увеличивайте `back_log` (подробнее см. в главах 8 и 9).
- *Остаются ли соединения открытыми даже тогда, когда не используются?* Если да, особенно при подключении сразу к нескольким серверам, то, возможно, вы используете соединения, которые могли бы пригодиться другим процессам. Пусть, например, приложение подключается к десяти серверам MySQL. Установить десять соединений из процесса Apache нетрудно, но лишь на одном из них в каждый момент времени происходит что-то полезное. Остальные девять просто проводят время в состоянии `Sleep`. Если один сервер начинает работать медленно или наблюдается длительный сетевой вызов, то остальные серверы могут испытывать нехватку соединений. Решение состоит в том, чтобы контролировать использование соединений приложением. Например, можно отправлять каждому экземпляру MySQL пакет запросов и закрывать соединение с ним перед тем, как обращаться к следующим. Если выполняется продолжительная операция, например обращение к веб-службе, то можно вообще закрыть соединение с MySQL, дождаться завершения запроса, а потом открыть соединение заново и продолжить работу с базой данных.

Различие между постоянными соединениями и пулом соединений может сбивать с толку. Постоянные соединения могут вызывать те же побочные эффекты, что и пул соединений, поскольку при повторном использовании соединения в любом случае сохраняется его состояние.

Однако пул обычно не приводит к открытию очень большого количества соединений, так как они используются несколькими процессами совместно. В то же время постоянные соединения создаются в рамках одного процесса и не разделяются с другими.

Кроме того, пул позволяет более точно управлять политикой создания соединений. Можно сконфигурировать пул так, что он будет автоматически увеличиваться, но обычно при исчерпании пула новые запросы ставятся в очередь. В этом случае запрос на открытие соединения ожидает на сервере приложений, а не перегружает сервер MySQL.

Существует много способов ускорить выполнение запросов и создание соединений, но общее правило состоит в том, что лучше не ускорять, а попытаться обойтись совсем без них.

Проблемы веб-сервера

Наиболее популярным сервером для создания веб-приложений является Apache. Он хорош для самых разных ситуаций, но при неправильном использовании может потреблять очень много ресурсов. Самые распространенные причины проблем состоят в том, что процессы живут слишком долго и используются для достижения комплекса целей. Лучше оптимизировать различные экземпляры для каждого типа задач.

Обычно Apache запускается с модулями `mod_php`, `mod_perl` и `mod_python` в конфигурации с заранее порожденными рабочими процессами. При этом каждому запросу назначается отдельный процесс. Поскольку скрипты на PHP, Perl и Python могут быть очень требовательны к ресурсам, нередко складывается ситуация, когда каждый процесс потребляет 50, а то и 100 Мбайт памяти. По завершении обработки запроса большая часть памяти возвращается операционной системе, но не вся. Apache не завершает процесс, а повторно использует его для последующих запросов. Таким образом, если следующим поступит запрос на статический ресурс, например CSS-файл или изображение, то этот простенький запрос будет обслуживаться очень «толстым» процессом. Именно поэтому так опасно использовать Apache в качестве универсального веб-сервера. Он действительно является универсальным, но специализация позволяет добиться более высокой производительности.

Еще одна серьезная проблема возникает из-за того, что процесс долго остается занятым, если включен режим Keep-Alive. Но даже если этот режим не включен, процесс может жить слишком долго, отдавая данные медленному клиенту в час по чайной ложке[1].

[1] Феномен кормления с ложечки возникает, когда клиент отправляет HTTP-запрос, но не забирает данные достаточно быстро. Пока клиент не получит все данные, HTTP-соединение, а значит, и процесс Apache остаются открытыми.

Часто администраторы совершают ошибку, оставляя включенным подразумеваемый по умолчанию набор модулей Apache. Размер памяти, занимаемый Apache, можно уменьшить, отключив ненужные модули. Сделать это просто: достаточно закомментировать лишние строки в конфигурационном файле Apache, а потом перезапустить его. Можно также удалить ненужные модули PHP из файла `php.ini`.

Подытоживая, повторим: если Apache запущен в универсальной конфигурации и напрямую обслуживает запросы, поступающие из Веб, то может быть порождено много тяжеловесных процессов. Это приводит к расточительному расходованию ресурсов веб-сервера. К тому же эти процессы могут открывать много соединений с MySQL, истощая и его ресурсы тоже. Приведем несколько рекомендаций, как уменьшить нагрузку на серверы[1].

- Не задействуйте Apache для обслуживания статического содержимого или по крайней мере используйте для этой цели отдельный экземпляр Apache. Популярными альтернативами являются Nginx (http://www.nginx.com) и lighttpd (http://www.lighttpd.net).
- Применяйте кэширующий прокси-сервер, например Squid или Varnish, чтобы вообще не допустить запросы до веб-сервера. Даже если на этом уровне невозможно кэшировать страницу целиком, иногда удается кэшировать большую ее часть и воспользоваться технологией edge side includes (ESI, см. http://www.esi.org) для включения небольшого динамического фрагмента в кэшированную статическую часть страницы.
- Задавайте срок хранения для статического и динамического содержимого. Кэширующий прокси-сервер, например Squid, позволяет явно сделать содержимое недействительным. На сайте «Википедии» эта техника применяется для удаления измененных статей из кэша.
- Иногда для того, чтобы увеличить срок хранения в кэше, требуется внести изменения в приложение. Например, если вы потребуете, чтобы браузер кэшировал CSS- и JavaScript-файлы вечно, а затем измените HTML-разметку сайта, страницы могут прорисовываться неправильно. Для борьбы с этим рекомендуется явно включать в имя файла номер версии. Например, скрипт публикации сайта можно написать так, чтобы CSS-файлы копировались с именем `/css/123_frontpage.css`, где `123` — номер редакции в системе Subversion. То же самое относится к файлам изображений — никогда не используйте повторно одни и те же имена, и после изменения графики страница будет отображаться правильно независимо от того, какой срок хранения задан в кэше браузера.
- Не позволяйте Apache кормить клиента с ложечки. Мало того что это медленно, так вы еще и распахиваете дверь для атак типа «отказ от обслуживания».

[1] Оптимизации веб-приложений посвящена хорошая книга Стива Соудерса (Steve Souders) High Performance Web Sites (издательство O'Reilly). Хотя в ней рассматривается главным образом создание быстрых с точки зрения клиента сайтов, предлагаемые рекомендации будут положительно влиять и на серверы. Следующая книга Стива Even Faster Web Sites также хороша.

Аппаратные балансировщики нагрузки обычно буферизуют ответы веб-сервера, поэтому Apache может завершить свою часть работы быстро, а уж балансировщик будет потихоньку отдавать клиенту содержимое из буфера. Можно также поставить перед приложением Nginx, Squid или Apache в режиме управления событиями.

- Включайте gzip-сжатие. Для современных процессоров эти затраты ничтожны, зато позволяют сильно сэкономить на трафике. Если вы хотите сэкономить циклы процессора, то можете кэшировать сжатые версии страниц и обслуживать их с помощью облегченного веб-сервера, такого как Nginx.
- Не включайте в Apache режим Keep-Alive для соединений на больших расстояниях, поскольку в этом случае процессы Apache будут существовать слишком долго. Лучше поручите заботу о режиме Keep-Alive прокси-серверу, который защищает Apache от клиента. Для соединений между Apache и прокси-сервером режим Keep-Alive вполне допустим, потому что прокси открывает всего несколько соединений для получения данных от Apache. Это различие иллюстрируется на рис. 14.1.

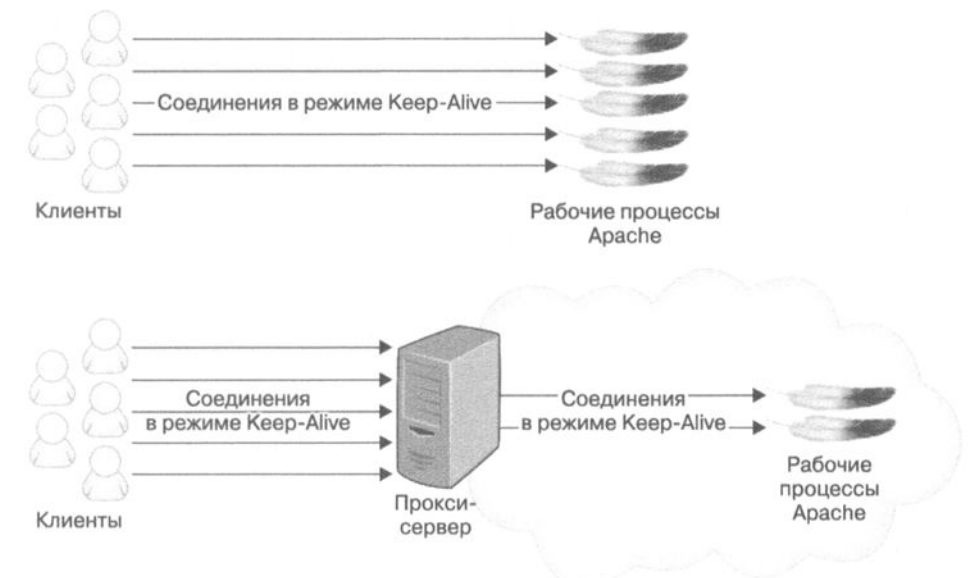


Рис. 14.1. Прокси-сервер может оградить Apache от долгоживущих соединений в режиме Keep-Alive, что позволяет уменьшить количество рабочих процессов Apache

Описанные приемы позволяют сократить время жизни процессов Apache, так что их будет не больше, чем действительно необходимо. Однако некоторые операции все же могут задержать процесс Apache надолго, потребляя при этом много ресурсов. Примером может служить запрос к внешнему ресурсу с большой задержкой, в частности к удаленному веб-сервису. Такие проблемы часто оказываются неразрешимыми.

В поисках оптимального уровня конкурентности. У каждого веб-сервера имеется оптимальный уровень конкурентности, то есть такое число конкурентных соединений, при котором запросы обрабатываются с максимальной скоростью, не перегружая систему. Это максимальная мощность системы, о которой мы говорили в главе 11. Для получения этого волшебного числа могут потребоваться небольшое измерение и моделирование или просто применение метода проб и ошибок, но результат того стоит.

Нередко бывает так, что сильно нагруженный веб-сервер одновременно обслуживает тысячи соединений, но лишь малая их часть занята активной обработкой данных. Остальные читают запросы, обеспечивают закачку файлов или просто ждут дальнейших запросов от клиента. По мере увеличения уровня конкурентности наступает момент, когда сервер достигает пика пропускной способности. После этого она перестает увеличиваться, а часто даже начинает снижаться. Но важнее то, что одновременно из-за очередей растет время отклика (задержка).

Чтобы понять, почему так происходит, предположим, что имеется один процессор и сервер одновременно обслуживает 100 запросов. На обслуживание каждого запроса процессор тратит 1 секунду. Предположим, что планировщик операционной системы работает вообще без накладных затрат и что на контекстное переключение тоже не расходуется время. Тогда для обработки всех запросов потребуется 100 секунд процессорного времени.

Как лучше всего обслуживать запросы? Можно выбирать их из очереди один за другим или обрабатывать параллельно, переключаясь с одного на другой и уделяя каждому из них одинаковое количество времени. В обоих случаях пропускная способность составляет один запрос в секунду. Однако средняя задержка при обработке по очереди (уровень конкурентности 1) составляет 50 секунд, а при параллельной (уровень конкурентности 100) — 100 секунд. На практике при параллельной обработке задержка будет даже выше из-за затрат на переключение контекста.

Если рабочая нагрузка ограничена процессорными ресурсами, то оптимальный уровень конкурентности равен количеству процессоров (или ядер). Однако процесс не готов к работе постоянно, поскольку он выполняет блокирующие системные вызовы, например запросы ввода/вывода, обращения к базе данных или сети. Поэтому, как правило, оптимальный уровень конкурентности несколько выше, чем количество процессоров.

Оценить оптимальный уровень конкуренции можно, но для этого потребуется точное профилирование. Обычно проще поэкспериментировать и посмотреть, при каком уровне конкурентности получается пиковая пропускная способность без замедления реакции, либо измерить и проанализировать реальную рабочую нагрузку. Инструмент pt-tcp-model пакета Percona Toolkit может помочь вам измерить и смоделировать масштабируемость и производительность вашей системы на основе содержимого памяти TCP.

Кэширование

Кэширование жизненно необходимо для высоконагруженных приложений. Типичное веб-приложение тратит на генерацию страницы гораздо больше ресурсов, чем на ее выборку из кэша (даже с учетом проверки и устаревания данных), поэтому обычно кэширование позволяет повысить производительность на несколько порядков. Сложность кроется в том, чтобы найти правильное сочетание детальности и правил истечения срока хранения. Кроме того, нужно решить, какое содержимое кэшировать и в каком месте.

В типичном высоконагруженном приложении существует много уровней кэширования. Оно производится не только на ваших серверах, но и на всем пути следования, включая браузер пользователя (именно для этой цели предназначены HTTP-заголовки, содержащие указания относительно срока хранения содержимого). В общем случае чем ближе кэш к клиенту, тем больше ресурсов он экономит и тем он эффективнее. Извлекать изображение из кэша браузера лучше, чем из памяти веб-сервера, а последнее лучше, чем читать его с диска сервера. Каждому типу кэша присущи свои характеристики, например размер и задержка. Некоторые из них мы рассмотрим в последующих разделах.

Кэши можно подразделить на две большие категории: *пассивные* и *активные*. Пассивный кэш только хранит и возвращает данные. Запрашивая что-то из пассивного кэша, вы получаете либо результат, либо сообщение «такого у меня нет». Примером пассивного кэша является memcached.

Напротив, активный кэш выполняет какие-то действия, когда искомое не обнаруживается. Обычно он передает поступивший запрос какой-то другой части приложения, которая и генерирует затребованный результат. Затем кэш сохраняет его и возвращает запросившей программе. Кэширующий прокси-сервер Squid — типичный пример активного кэша.

При проектировании приложения обычно стремятся сделать кэши активными (их еще называют *прозрачными*), поскольку они скрывают от программного обеспечения логику проверки, генерации и сохранения. Активные кэши можно строить поверх пассивных.

Кэширование на уровне ниже приложения

У сервера MySQL есть свои внутренние кэши, да и вам никто не запрещает создать собственные кэшированные и сводные таблицы. Кэшированные таблицы можно спроектировать так, чтобы они повышали скорость фильтрации, сортировки, соединения с другими таблицами и решения других специализированных задач. К тому же кэшированные таблицы существуют дольше, чем большинство кэшей на уровне приложения, поскольку они не исчезают при перезапуске сервера.

Мы писали об этих стратегиях кэширования в главах 4 и 5, поэтому здесь сфокусируемся на кэшировании на уровне приложения и выше.

Кэширование не всегда помогает

Нужно убедиться, что в результате кэширования производительность действительно повышается, потому что иногда кэш не помогает вовсе. Например, на практике часто быстрее обслужить запрос из памяти Nginx, чем из кэша прокси-сервера на диске.

Причина проста: с кэшированием связаны определенные накладные затраты. Необходимо проверить наличие данных в кэше и, если это так, обслужить запрос оттуда. Чтобы поместить данные в кэш или сделать их недействительными, тоже требуются накладные затраты. Кэширование полезно лишь в том случае, когда они не превышают затрат на генерацию и обслуживание страницы без кэша.

Если вы знаете, каковы затраты на все эти операции, то можно подсчитать, в какой мере кэш окажется полезен. Стоимость обслуживания без кэша — это затраты на генерацию данных при каждом запросе. Стоимость обслуживания с кэшем равна затратам на проверку наличия данных в кэше плюс вероятность непопадания, умноженная на затраты на генерацию данных, плюс вероятность попадания, умноженная на затраты на обслуживание данных из кэша.

Если стоимость при использовании кэша меньше стоимости без его применения, то производительность может повыситься, хотя это и не гарантируется. Также имейте в виду, что, как и в случае обслуживания данных из памяти Nginx, а не из кэша на прокси-сервере, некоторые кэши менее затратны, чем другие.

Кэширование на уровне приложения

Кэш уровня приложения обычно размещается в памяти на том же самом компьютере или в памяти другого компьютера, доступного по сети.

Кэширование на уровне приложения может оказаться более эффективным, чем на более низких уровнях, поскольку позволяет приложению сохранять в кэше частично вычисленные результаты. Поэтому кэш избавляет ПО от двух видов деятельности: выборки данных и вычислений с ними. Хороший пример показывают блоки HTML-разметки. Приложение может сгенерировать HTML-фрагменты, например заголовки новостей, и поместить их в кэш. При последующих просмотрах страницы нужно будет лишь вставить в нее кэшированный текст. В общем случае чем глубже обработка данных перед кэшированием, тем больше времени удается сэкономить при попадании в кэш.

Недостаток состоит в том, что коэффициент попадания в кэш может оказаться низким, а сам кэш будет занимать больше памяти. Предположим, нужны 50 вариантов новостных заголовков, чтобы пользователь видел те, которые относятся к местности, где он проживает. Тогда кэш займет память, необходимую для хранения всех вариантов, но на каждый вариант будет приходиться меньше попаданий, да и логика удаления из кэша усложнится.

Существует много разных видов кэшей на уровне приложения. Опишем некоторые из них.

- *Локальные кэши.* Обычно невелики и существуют только в памяти процесса на протяжении обработки одного запроса. Они полезны, когда нужно избежать повторного обращения к ресурсу, который необходим более одного раза. В таком кэше нет ничего хитроумного, обычно это просто переменная или хеш-таблица в коде приложения. Пусть, например, нужно вывести имя пользователя, зная его идентификатор. Можно написать функцию `get_name_from_id()` и реализовать в ней кэширование, как показано далее:

```
<?php
function get_name_from_id($user_id) {
    static $name; // ключевое слово static сохраняет
                  // значение переменной между вызовами
    if ( !$name ) {
       // Выбрать имя из базы данных
    }
    return $name;
}
?>
```

  На языке Perl стандартный способ кэширования результатов вызова функций дает модуль `Memoize`:

```
use Memoize qw(memoize);
memoize 'get_name_from_id';
sub get_name_from_id {
   my ( $user_id ) = @_;
   my $name = # get name from database
   return $name;
}
```

  Эти приемы просты, но позволяют избавить приложение от лишней работы.

- *Кэши в локальной разделяемой памяти.* Это кэши среднего размера — до нескольких гигабайтов, они работают быстро, но с трудом поддаются синхронизации между несколькими машинами. Хороши для небольших, полустатических фрагментов данных. В качестве примеров можно привести списки городов штатов, функцию секционирования (таблицу отображения) для шардированного хранилища данных или значения, которые делаются недействительными по истечении времени жизни (time-to-live, TTL). Основное достоинство разделяемой памяти — очень высокая скорость доступа, обычно она гораздо выше, чем для любого удаленного кэша.

- *Распределенные кэши в памяти.* Самый известный пример такого рода — memcached. Объем распределенных кэшей гораздо больше, чем кэша в локальной разделяемой памяти, и их размер легко увеличивается. В кэше создается всего одна копия данных, поэтому не нужно тратить время и решать проблемы согласования, возникающие, когда одна и та же информация кэшируется в разных местах. Распределенная память прекрасно подходит для хранения таких разделяемых объектов, как профили пользователей, комментарии и фрагменты HTTP-разметки.

Но время задержки у таких кэшей гораздо выше, чем у кэшей в локальной разделяемой памяти, поэтому наиболее эффективный способ их использования — получение сразу нескольких объектов за одно обращение. Кроме того, вам следует продумать, как будут добавляться новые узлы и что делать, если какой-нибудь узел откажет. В обеих ситуациях приложение должно решить, как распределять или перераспределять объекты между узлами.

Согласованность кэширования важна в случае, когда в кластер кэшей добавляется новый сервер или какой-то сервер из кластера удаляется. Для memcached, находящегося по адресу http://www.audioscrobbler.net/development/ketama/, имеется библиотека обеспечения согласованности.

- *Кэши на диске.* Диски — медленные устройства, поэтому на них лучше кэшировать объекты, не помещающиеся в памяти, или статическое содержимое, например предварительно сгенерированные изображения.

  Очень полезный прием, применяемый в случае кэширования на диске в контексте веб-сервера, состоит в том, чтобы написать обработчик ошибки 404, который будет перехватывать непопадания в кэш. Предположим, что веб-приложение показывает в заголовке генерируемое на лету изображение, содержащее имя пользователя («С возвращением, Джон!»). На это изображение можно сослаться по адресу `/images/welcomeback/john.jpg`. Если его еще не существует, то возникнет ошибка 404 и будет вызван ее обработчик. Он сгенерирует изображение, поместит его на диск и либо выполнит перенаправление, либо просто вернет браузеру поток байтов, представляющий собой изображение. Все последующие запросы будут просто возвращать картинку из файла.

  Этот прием годится и для многих других типов содержимого. Например, вместо того, чтобы кэшировать заголовки последних новостей в виде блоков HTML, можно сохранить их в виде JavaScript-файла и затем сослаться на этот файл `/latest_headlines.js` из заголовка веб-страницы.

  Сделать элемент кэша недействительным очень просто: достаточно удалить файл. Чтобы реализовать удаление по истечении времени жизни, можно периодически запускать задание, которое будет стирать файлы, созданные более *N* минут назад. А если требуется ограничить размер кэша, то можно применить политику вытеснения по давности использования (least recently used, LRU) элементов, удаляя файлы в порядке времени последнего доступа.

  Для того чтобы вытеснение по времени последнего доступа работало, необходимо включить в параметрах монтирования файловой системы режим регистрации данного события (нужно лишь снять флаг `noatime`). Но если вы решите так поступить, то следует задействовать файловую систему в памяти, иначе диск будет использоваться очень активно.

Стратегии управления кэшем

Кэшам присуща та же проблема, что и денормализованной базе данных: значения дублируются, то есть информацию приходится обновлять в нескольких местах и необходимо принимать меры к тому, чтобы избежать чтения некорректных дан-

ных. Далее описываются некоторые наиболее употребительные стратегии управления кэшем.

- *TTL (время жизни).* Вместе с кэшированным объектом хранится момент истечения срока хранения — можно либо удалять все объекты, для которых этот момент уже наступил, с помощью специального процесса, либо дождаться следующего обращения и заменить объект более актуальной версией. Такая стратегия вытеснения больше подходит для редко изменяющихся данных или таких, для которых требования к актуальности некритичны.
- *Явная инвалидация.* Если кэширование устаревших данных неприемлемо, то процесс, который обновляет исходную информацию, может делать недействительной версию в кэше. У такой стратегии есть два варианта: *инвалидировать при записи* и *обновить при записи.* Стратегия инвалидации при записи проста: нужно просто пометить данные в кэше флажком, показывающим, что срок хранения истек и можно удалить эти данные из кэша. Обновление при записи требует чуть больше работы, поскольку нужно заменить устаревший элемент в кэше обновленным. Но это может дать заметный выигрыш, особенно в тех случаях, когда генерировать кэшируемые значения затратно, а процесс, осуществляющий обновление, возможно, их уже имеет. Если данные в кэше обновлены, то последующие запросы не должны будут ждать, пока приложение сгенерирует их. Если инвалидация производится в фоновом режиме (такова инвалидация по времени жизни), то новые версии удаленных элементов можно сгенерировать в процессе, никак не связанном с запросами пользователей.
- *Инвалидация при чтении.* Вместо того чтобы инвалидировать устаревшие данные из кэша в момент изменения исходных значений, можно сохранить информацию, которая позволит определить, истек ли срок хранения данных, в момент чтения их из кэша. По сравнению с явным инвалидированием у этой стратегии есть важное преимущество: фиксированные затраты на операцию распределяются по времени. Предположим, что изменился объект, от которого зависят миллион других объектов в кэше. Если применять стратегию инвалидации при записи, то придется за один раз сделать недействительными миллион кэшированных элементов, что может занять длительное время, даже если вы придумаете эффективный алгоритм поиска таких элементов. Если же производить инвалидацию при чтении, то запись можно выполнить сразу, а миллион операций чтения отложить на потом. Таким образом, затраты на вытеснение распределяются по времени и нам удается избежать пиков нагрузки и больших задержек.

Один из самых простых подходов к реализации инвалидации при чтении — отслеживание версий объектов. В этом случае вместе с объектом в кэше хранится номер текущей версии или временная метка данных, от которых зависит кэшированный объект. Пусть, например, кэшируется статистика блога пользователя, в том числе количество отправленных им сообщений. Вместе с объектом `blog_stats` в кэш помещается номер текущей версии пользователя, поскольку статистика зависит именно от него.

При каждом обновлении любых данных, которые также зависят от этого пользователя, обновляется номер его версии. Предположим, что начальный номер версии — 0, и в этот момент статистика генерируется и сохраняется в кэше. Когда пользователь

публикует в своем блоге новое сообщение, номер версии увеличивается на 1 (его следовало бы сохранить и в самом сообщении, хотя для рассматриваемого примера это не обязательно). Позже при отображении статистики мы сравниваем версию объекта `blog_stats` в кэше с версией пользователя. Поскольку номер версии пользователя больше, то мы знаем, что статистика устарела и должна быть вычислена заново.

Это довольно грубый способ инвалидации содержимого, поскольку предполагается, что любые данные, как-то зависящие от пользователя, влияют на любые другие данные. Не всегда это так. Например, если пользователь редактирует свое сообщение, то номер его версии будет изменен, что приведет к инвалидации статистики, хотя она (количество сообщений в блоге) и не изменилась. В обмен на такую неточность мы получаем простоту. Незамысловатую стратегию инвалидации не только проще реализовать, не исключено, что и работать она будет эффективнее.

Отслеживание версий объектов — это упрощенный вариант тегированного кэша, который может учитывать и более сложные зависимости. Тегированный кэш знает о разнообразных зависимостях и для каждой из них поддерживает самостоятельные версии. Возвращаясь к примеру с клубом книголюбов из главы 11, можно было бы кэшировать комментарии в зависимости от версии пользователя и версии книги. Для этого нужно было бы снабдить комментарий тегом, состоящим из двух номеров версий: `user_ver=1234` и `book_ver=5678`. Если хотя бы одна версия изменится, то нужно будет обновить комментарий в кэше.

Кэширование иерархий объектов

Иерархическое хранение объектов в кэше может упростить выборку и инвалидацию и помочь более эффективно использовать память. Вместо того чтобы кэшировать только сами объекты, можно кэшировать их идентификаторы, а также группы идентификаторов объектов, которые обычно извлекаются совместно.

Хороший пример применения такой техники дает сайт электронной торговли. Результатом поиска может быть список подходящих товаров, включающий наименования, описания, миниатюры фотографий и цены. Кэшировать список целиком было бы неэффективно: вполне вероятно, что другой поиск вернет некоторые из уже найденных товаров, так что мы будем хранить дубликаты, впустую транжиря память. Более того, при такой стратегии было бы сложно найти и сделать недействительными те из кэшированных результатов поиска, которые содержат товары с изменившейся ценой, поскольку для этого пришлось бы просматривать все списки без исключения.

Вместо того чтобы кэшировать перечень товаров полностью, можно сохранить лишь минимальную информацию о поиске, например количество найденных элементов и список их идентификаторов. А каждый товар будет кэшироваться по отдельности. Тем самым мы решаем обе проблемы: результаты не дублируются и вытеснение можно производить с точностью до отдельного товара.

Недостаток заключается в том, что из кэша приходится извлекать несколько объектов, а не один полный результат поиска. Однако сохранение списка идентификаторов делает эту процедуру довольно эффективной. При попадании в кэш возвращается список идентификаторов, который затем используется для повторного обращения к кэшу. Результатом второго обращения может быть группа товаров, если кэш позволяет получать несколько объектов за одно обращение (memcached поддерживает такую возможность посредством функции `mget()`).

Однако, если вы будете невнимательны, этот подход может дать странные результаты. Допустим, применяется стратегия инвалидации результатов поиска по времени жизни, а объекты, описывающие отдельные товары, вытесняются при изменении. Представьте, что произойдет, когда описание товара изменится так, что перестанет содержать ключевые слова, отвечающие условиям поиска, однако результат поиска еще не удален из кэша. Пользователи увидят устаревшие данные, поскольку кэшированный список идентификаторов ссылается на товар, который больше не удовлетворяет заданным критериям поиска.

Для большинства приложений это не слишком серьезная проблема. Но если для вашего ПО такая ситуация неприемлема, то можно воспользоваться кэшированием с версиями и хранить вместе с результатами поиска версии товаров. Обнаружив поисковый результат в кэше, вы можете сравнить версию товара в нем с текущим (кэшированным) параметром. Если какой-нибудь товар устарел, можно повторить поиск и заново кэшировать результаты.

Важно понимать, насколько затратен доступ к удаленно хранимому кэшу. Хотя кэши бывают быстрыми и не требуют больших объемов работы, сетевое взаимодействие на сервере кэширования в локальной сети обычно занимает около 0,3 миллисекунды. Мы встречали немало случаев, когда для сборки сложной веб-страницы требовалось выполнить около 1000 запросов к кэшу. В итоге это составляет 3 секунды сетевой задержки, что означает: ваша страница может загружаться недопустимо медленно, даже если она обслуживается без обращений к базе данных! Использование вызова, позволяющего получать много объектов за одно обращение к кэшу, в такой ситуации необходимо. Использование иерархии кэша с небольшим локальным кэшем также может быть очень полезным.

Предварительная генерация содержимого

Помимо кэширования данных на уровне приложения, можно заранее запрашивать некоторые страницы из фоновых процессов и сохранять полученные результаты в виде статических страниц. Если страницы формируются динамически, то можно заранее сгенерировать какие-то их части, а для построения полной страницы применить технологию включения на стороне сервера. Это позволит сократить размер предварительной генерации содержимого и затраты на нее, поскольку в противном случае придется дублировать значительные объемы данных из-за мелких

различий в порядке сборки страницы из отдельных частей. Стратегию предварительной генерации можно использовать почти для любого типа кэширования, включая и memcached.

Предварительная генерация содержимого обладает несколькими важными достоинствами.

- Код приложения становится проще, поскольку не учитывает ситуации попаданий и непопаданий в кэш.
- Эта стратегия работает хорошо, когда операции, следующие за непопаданием в кэш, являются неприемлемо медленным, поскольку она гарантирует, что непопаданий никогда не будет. Фактически при проектировании системы кэширования любого типа вы всегда должны учитывать, насколько медленно выполняются операции, если попадания в кэш не произошло. Если средняя производительность значительно возрастает, но случайный запрос становится чрезвычайно медленным из-за восстановления кэшированного содержимого, то кэширование может оказаться даже худшей стратегией, чем отказ от использования кэша. Постоянная величина производительности часто так же важна, как и высокая производительность.
- Предварительная генерация содержимого не вызывает бегство кэша в случае непопадания.

Для кэширования предварительно сгенерированного содержимого может потребоваться много места, и не всегда возможно создать все страницы. Но, как обычно при кэшировании, важно заранее подготовить то содержимое, которое запрашивается чаще всего, поэтому можно применить технику генерации по необходимости с помощью обработчиков ошибки 404, которую мы упоминали в этой главе.

Иногда можно получить выигрыш, организуя хранение предварительно сгенерированного содержимого в файловой системе, размещенной в памяти, чтобы избежать дискового ввода/вывода.

Кэш как инфраструктурный компонент

Скорее всего, кэш будет важной частью вашей инфраструктуры. Легко попасть в ловушку, думая о кэше как о некоем удобстве, которое неплохо иметь, но не о том, что настолько важно, что вы не можете жить без него. Возможно, вы считаете, что если сервер кэша выйдет из строя или вы потеряете кэшированное содержание, запрос просто будет перенаправлен в базу данных. Так и случится, когда вы первоначально добавляете кэш в приложение, но кэш может позволить приложению значительно расти без увеличения ресурсов, предназначенных для некоторой части системы, как правило базы данных. В результате вы можете стать зависимыми от кэша, даже не осознавая этого.

Например, если коэффициент попадания в кэш составляет 90 % и по какой-то причине вы кэш потеряете, время загрузки базы данных увеличится в десять раз. Скорее всего, это превысит мощность сервера базы данных.

Чтобы избежать таких неожиданностей, следует подумать о каком-то решении с высокой доступностью для кэша (как данных, так и службы) или по крайней мере измерить влияние отключения кэша или потери содержащихся в нем данных на производительность. Например, вам может потребоваться разработать приложение для ухудшения его функциональности.

Использование HandlerSocket и доступа к memcached

Вместо хранения данных в MySQL и кэширования их вне MySQL можно предусмотреть более быстрый путь для доступа к MySQL и последующего игнорирования кэша. При небольших простых запросах бóльшая часть накладных затрат может возникнуть в результате разбора SQL, проверки привилегий, создания плана выполнения и т. д. Если они будут устранены, при простых запросах MySQL может оказаться очень быстрой.

В настоящее время есть два решения, которые используют эту ситуацию, позволяя так называемой NoSQL получить доступ к MySQL. Первым является служебный плагин под названием HandlerSocket, который был создан на DeNA — крупном сайте японской социальной сети. Он позволяет вам получить доступ к объекту InnoDB Handler через простой протокол. По сути, вы проходите мимо верхних уровней сервера и напрямую подключаетесь к InnoDB по сети. Имеются сообщения о том, что HandlerSocket выполняет более 750 000 запросов в секунду. HandlerSocket распространяется вместе с Percona Server, а memcached-доступ к InnoDB включен в тестовый релиз MySQL 5.6.

Второй способ — доступ к InnoDB через протокол memcached. В лабораторных релизах MySQL 5.6 есть плагин, который позволяет это.

Оба подхода несколько ограничены, особенно подход через memcached, который не поддерживает столько методов доступа к данным. Зачем бы вам хотеть получить доступ к данным через что-либо еще помимо SQL? Самая важная, помимо скорости, причина — вероятно, простота. Избавиться от кэша и всей сопутствующей логики инвалидации и дополнительной инфраструктуры — это большая победа.

Расширение MySQL

Если MySQL не умеет делать того, что вам требуется, можно попытаться расширить ее возможности. Мы не станем показывать, как это делается, но упомянем некоторые способы. Если вы хотите пойти по этому пути, то в Сети найдете немало отличных сетевых ресурсов и книг, в которых рассматриваются многие из затронутых здесь тем.

Говоря: «MySQL не умеет делать того, что вам требуется», мы обычно имеем в виду две вещи: MySQL вообще не умеет этого делать или умеет, но недостаточно быстро или слишком неуклюже. В каждом из этих случаев имеет смысл подумать о расширении. К счастью, MySQL становится все более и более модульной и универсальной системой.

Подсистемы хранения — отличный способ расширить MySQL для решения узкоспециализированной задачи. Брайан Эйкер (Brian Aker) написал заготовку подсистемы хранения и опубликовал серию статей и презентаций, посвященных началу разработки собственной подсистемы. Они послужили основой для создания нескольких значимых подсистем хранения от сторонних производителей. Ныне многие компании пишут собственные подсистемы хранения для внутренних целей. Например, некоторые социальные сети применяют специальную подсистему хранения для работы с графами, описывающими отношения между людьми. Нам известна также компания, создавшая подсистему хранения для нечеткого поиска. Простенькую подсистему хранения написать не так уж трудно.

Кроме того, подсистему хранения можно использовать в качестве интерфейса для другого программного обеспечения. Хорошим примером является подсистема хранения Sphinx, которая взаимодействует с программным обеспечением полнотекстового поиска Sphinx (см. приложение E).

Альтернативы MySQL

MySQL не может удовлетворить все мыслимые потребности. Порой гораздо лучше решать некую задачу вне MySQL, даже если теоретически она способна с ней справиться.

Один из самых очевидных примеров — хранение данных в традиционной файловой системе, а не в таблицах. Классический случай — файлы изображений: их можно поместить в столбец типа `BLOB`, но это редко бывает оправданно[1]. Обычно изображения и другие большие двоичные файлы размещают в файловой системе, а внутри MySQL хранят только их имена, и приложение может получить файл в обход MySQL. В веб-приложении это достигается записью имени файла в атрибут `src` тега `<img>`.

Полнотекстовый поиск также лучше реализовывать внешними средствами, поскольку MySQL не способна делать это так же хорошо, как, скажем, системы Lucene или Sphinx.

Для некоторых задач полезен NDB API. Например, хотя подсистема хранения NDB Cluster пока непригодна для хранения всех данных в высокопроизводительном веб-приложении, ее все же можно использовать в качестве хранилища информации о сеансах или регистрационных данных пользователей. Подробнее о NDB API можно прочитать на странице http://dev.mysql.com/doc/ndbapi/en/index.html. Существует также mod_ndb — модуль NDB для Apache, который можно скачать с сайта http://code.google.com/p/mod-ndb/.

Наконец, для некоторых операций, например обхода деревьев и работы с графами, реляционная база данных просто не предназначена. MySQL плохо приспособлена

[1] Один из случаев, когда хранение файлов в MySQL оправданно, — это использование репликации для быстрой доставки изображений на множество машин. И мы знаем несколько приложений, в которых такая техника применяется.

для взаимодействия с распределенными данными, поскольку ей недостает средств распараллеливания обработки запросов. По-видимому, для этой цели стоит поискать другие инструменты (возможно, в сочетании с MySQL). На ум приходят такие примеры.

- Мы заменили MySQL на Redis, в котором простые пары «ключ — значение» обрабатывались с такой высокой скоростью, что подчиненные серверы отстали, хотя главный сервер справлялся с нагрузкой просто отлично. Redis также популярен для очередей благодаря хорошей поддержке операций в очереди.
- И еще Hadoop — слона-то вы приметили, правда? Гибридные варианты развертывания MySQL/Hadoop очень распространены для обработки больших или полуструктурированных наборов данных.

Итоги главы

Оптимизация — это задача не только базы данных. Как мы предположили в главе 3, лучшая форма оптимизации ориентирована как на бизнес-логику, так и на удобство пользователя. Оптимизация производительности во всем стеке — то, что действительно необходимо для достижения этой цели.

Первое, что нужно сделать, — как всегда, выполнить измерения. Сосредоточьтесь на профилировании, поочередно рассматривая слои. На каких слоях на отклик затрачивается больше всего времени? Определите их. Если удобство работы с браузером зависит в основном от того, как в нем отображается модель DOM, а быстродействие MySQL незначительно сказывается на общем времени отклика, то дальнейшая оптимизация запросов практически ничем не поможет пользователю. После измерений, как правило, становится очевидно, куда следует направить основные усилия. Мы рекомендуем прочитать книгу Стива Судерса (Steve Souders) *High Performance Web Sites and Even Faster Web Sites* и использовать New Relic.

Часто проблему можно решить конфигурацией веб-сервера и кэшированием. Согласно стереотипному представлению, «проблема всегда с базой данных», но это просто неверно. Другие слои в приложении не менее важны, и их точно так же можно неправильно сконфигурировать, хотя иногда эффект от этого менее очевиден. В частности, кэш может помочь получить много контента с гораздо меньшими затратами, чем это можно было бы сделать только с MySQL. Несмотря на то что Apache по-прежнему является самым популярным в мире программным обеспечением для веб-серверов, это не всегда наилучший инструмент для работы, поэтому в некоторых ситуациях имеет смысл обратить внимание на альтернативы, например Nginx.

15 Резервное копирование и восстановление

Если вы не запланировали резервное копирование на ранних этапах проекта, то потом можете обнаружить, что отказались от полезного действия. Например, вы уже сконфигурировали сервер, а затем осознали, что надо бы воспользоваться LVM для создания мгновенных снимков файловой системы, но уже было поздно. Кроме того, может выясниться, что настройка системы для резервного копирования серьезно влияет на производительность. И если заранее не сформировать план восстановления и не испытать его на практике, то в момент аварии все пойдет совсем не так гладко, как хотелось бы.

В отличие от авторов первого и второго изданий этой книги мы полагаем, что большинство читателей используют InnoDB, а не MyISAM. Мы не собираемся рассматривать в этой главе все компоненты хорошо спроектированного решения по резервному копированию и восстановлению — только части, относящиеся к MySQL. Вот несколько тем, которые мы решили в нее не включать.

- Безопасность (доступ к резервному копированию, привилегии для восстановления данных, должны ли файлы быть зашифрованы).
- Где хранить резервные копии, в том числе как далеко от источника они должны находиться (на другом диске, на другом сервере или за его пределами) и как перемещать данные из источника в пункт назначения.
- Политика хранения, аудит, требования законодательства и связанные с этим темы.
- Системы хранения и носители, сжатие и инкрементное копирование.
- Форматы хранения данных.
- Мониторинг и отчеты о резервном копировании.
- Встроенные в уровни хранения возможности резервного копирования или специальные устройства, такие как готовые аппаратные файловые серверы.

Эти темы раскрыты в книгах, например в *Backup & Recovery*, написанной В. Кёртисом Престоном (W. Curtis Preston) (издательство O'Reilly).

Прежде всего уточним основные термины. Во-первых, вы часто услышите о так называемом *горячем*, *теплом* и *холодном* резервном копировании. Обычно этими словами описывается влияние резервного копирования на текущую работу. Например, горячее резервное копирование означает, что сервер не нужно останавливать. Проблема в том, что не все одинаково понимают эти термины. В названиях некоторых инструментов даже присутствует слово hot (горячий), хотя они определенно не выполняют то, что мы понимаем под горячим резервным копированием. Мы постараемся избегать таких понятий, а вместо этого будем говорить о том, в какой мере конкретная методика или инструмент прерывают работу сервера.

Еще два сбивающих с толку слова — restore (возвращать) и recover (восстанавливать). В этой главе им придается вполне конкретный смысл. Под *возвратом* мы понимаем извлечение данных из резервной копии и либо загрузку их в MySQL, либо размещение файлов там, где MySQL ожидает их найти. Под *восстановлением* обычно понимается весь процесс приведения системы или ее части в рабочее состояние после того, как что-то пошло не так. Сюда входит не только возврат данных из резервной копии, но и прочие шаги, необходимые для возобновления функционирования в полном объеме, в частности перезапуск MySQL, изменение конфигурации, прогрев кэшей сервера и т. д.

Для многих восстановление означает лишь исправление поврежденных в результате сбоя таблиц. Но это совсем не то же самое, что восстановление всего сервера. Процедура восстановления подсистемы хранения должна привести в соответствие файлы данных и журналов. Она также должна гарантировать, что в файлах данных отражены только модификации, произведенные закоммиченными транзакциями, и повторить те транзакции из журналов, которые еще не были применены к файлам данных. Это может быть частью общего процесса восстановления или даже частью резервного копирования. Однако это не то восстановление, которое может потребоваться после случайного выполнения команды `DROP TABLE`.

Зачем нужно резервное копирование

Приведем несколько причин необходимости резервного копирования.

- *Аварийное восстановление.* Аварийное восстановление — это то, что требуется, когда аппаратное обеспечение выходит из строя, ужасная ошибка повреждает ваши данные или сервер и его данные становятся недоступными или непригодными для использования по какой-то другой причине. Вы должны быть готовыми ко всему: от того, что кто-то случайно подключился не к тому серверу и выполнил команду `ALTER TABLE`[1], до пожара в здании, от атаки хакера до ошибки MySQL. Пусть вероятности конкретных аварий не очень высоки, но они имеют тенденцию суммироваться.

[1] Бэрон все еще помнит свою первую после колледжа работу: он удалил два столбца из таблицы счетов рабочего сервера на сайте, занимающемся электронной торговлей.

- *Изменение решения*. Вы бы удивились, узнав, как часто люди намеренно удаляют данные, а затем хотят их восстановить.
- *Аудит*. Иногда вам требуется узнать, как выглядели данные или схема в какой-то момент времени в прошлом. Например, вы вовлечены в судебный процесс или обнаружили ошибку в приложении и хотите понять, что было написано в этом коде раньше (иногда того, что ваш код находится в системе контроля версий, недостаточно).
- *Тестирование*. Одним из самых простых способов тестирования на реальных данных является периодическое обновление тестового сервера с использованием последних рабочих данных. Если вы делаете резервные копии, это легко — просто восстановите резервную копию на тестовом сервере.

Проверьте свои предположения. Например, полагаете ли вы, что ваш хостинг-провайдер поддерживает резервную копию сервера MySQL, предоставляемого вместе с вашим аккаунтом? Одни хостинг-провайдеры не делают резервных копий серверов MySQL, а другие просто копируют файлы во время работы сервера, что, вероятно, создает поврежденную резервную копию, которая может оказаться бесполезной.

Определение требований к восстановлению

Пока все идет хорошо, о восстановлении как-то не задумываешься. Но когда момент настанет, не спасет даже самая лучшая в мире система резервного копирования. Вместо этого понадобится хорошая система восстановления.

Проблема в том, что организовать гладкий процесс резервного копирования проще, чем создать хорошие процедуры и инструменты восстановления. И вот почему.

- Резервное копирование происходит раньше. Если нет резервной копии, то восстанавливать будет не с чего, поэтому при построении системы внимание уделяют прежде всего резервному копированию.
- Резервное копирование автоматизировано с помощью скриптов и заданий. Поэтому вы думаете главным образом о том, как автоматизировать и настроить именно его, часто даже не отдавая себе в этом отчета. Пять минут в день на улучшение процедуры копирования — вроде бы пустяк, но уделяете ли вы ежедневно такое же внимание восстановлению?
- Снятие резервной копии обычно происходит без всякого стресса, а восстановление — как правило, в кризисной ситуации.
- Часто в дело вступают соображения безопасности. Если резервное копирование производится за пределами рабочей площадки, то, скорее всего, данные в копии шифруются или защищаются как-то иначе. Очень легко рассуждать об ущербе, который нанесет компрометация данных, и совсем упустить из виду то, что произойдет, если никто не сможет прочитать зашифрованный том, чтобы восстановить с него данные, или позабыть о том, чего стоит извлечь один файл из монолитного зашифрованного файла-архива.

- Один человек может спланировать, спроектировать и реализовать процедуру резервного копирования. Но его может не оказаться на месте, когда грянет гром. Необходимо обучить нескольких человек и позаботиться о том, чтобы кто-нибудь из них постоянно присутствовал, но не поручать восстановление данных неквалифицированному сотруднику.

Приведем случай из практики: некий заказчик сообщил, что резервное копирование стало выполняться с молниеносной скоростью, стоило указать при запуске mysqldump флаг `-d`, и поинтересовался, почему никто не сообщил ему, что этот флаг может настолько ускорить процедуру. Если бы он попытался вернуть данные с таких копий, то очень быстро понял бы причину: при наличии флага `-d` данные вообще не копируются! Заказчика интересовало только резервное копирование, а не восстановление, поэтому он и не ведал о грозящей беде.

Существуют два Больших Важных Требования, которые могут оказаться полезными при планировании стратегии резервного копирования и восстановления. Это *целевая точка восстановления* (recovery point objective, RPO) и *целевое время восстановления* (recovery time objective, RTO). Они определяют, сколько данных вы готовы потерять и как долго готовы ждать их восстановления. При определении своих RPO и RTO попытайтесь ответить на следующие типы вопросов.

- Сколько данных вы можете потерять без серьезных последствий? Понадобится ли восстановление на конкретный момент времени в прошлом или допустимо потерять всю работу с момента снятия последней резервной копии? Существуют ли какие-то законодательные требования?
- Насколько быстрым должно быть восстановление? Каково допустимое время простоя? С чем будут готовы смириться приложение и пользователи (например, с частичной недоступностью) и как вы собираетесь обеспечить продолжение функционирования в таких условиях?
- Что необходимо восстанавливать? Обычно речь идет обо всем сервере, об отдельной базе данных, отдельной таблице или о конкретных транзакциях и командах.

Задокументируйте ответы на эти вопросы, а также всю политику резервного копирования и его процедуры.

Миф о резервном копировании № 1: я использую репликацию как резервное копирование

Эта ошибка нам встречалась очень часто. Подчиненный сервер не может быть заменой резервной копии. И RAID-массив тоже. Чтобы понять почему, задумайтесь над таким вопросом: помогут ли они восстановить данные после случайного выполнения команды `DROP DATABASE` для промышленной базы данных? Ни RAID-массив, ни репликация не пройдут этого простейшего теста. Их нельзя относить к технологиям резервного копирования. Они даже не могут заменить резервную копию! Ничто, кроме самого резервного копирования, не способно реализовать функции резервного копирования.

Проектирование резервного копирования в MySQL

Резервное копирование MySQL сложнее, чем кажется. Прежде всего, резервная копия — это лишь копия данных, но потребности вашего приложения, архитектура подсистемы хранения MySQL и конфигурация системы могут затруднить ее создание.

Прежде чем подробно описывать существующие варианты, дадим некоторые рекомендации.

- Для больших баз данных физическое резервное копирование просто необходимо: логическое копирование выполняется очень медленно, потребляет много ресурсов, а восстановление с копии выполняется долго. Самые лучшие варианты — это резервное копирование на основе снимков, применение Percona XtraBackup и MySQL Enterprise Backup. Для маленьких баз данных логическое копирование работает вполне нормально.
- Сохраняйте несколько поколений резервных копий.
- Периодически создавайте логические резервные копии (возможно, из физических копий).
- Храните двоичные журналы для восстановления на определенный момент времени. Установите довольно большое значение параметра `expire_logs_days`, чтобы можно было восстановить минимум два поколения исходных резервных копий и существовала возможность создать подчиненный сервер и запустить его из работающего главного без применения двоичных журналов.

  Создайте резервные копии двоичных журналов, не зависящие от заданного срока хранения, и храните их в резервной копии так долго, чтобы была возможность восстановления по крайней мере из последней логической резервной копии.
- Ведите мониторинг процесса резервного копирования и самих резервных копий независимо от тех инструментов, которые применяются для резервного копирования. Необходимо независимое подтверждение их пригодности.
- Периодически проверяйте процедуру резервного копирования и восстановления, проходя процесс восстановления от начала до конца. Измеряйте ресурсы, необходимые для восстановления (процессор, объем диска, физическое время, полоса сети и т. п.).
- Не упускайте из виду безопасность. Что случится, если кто-то скомпрометирует сервер, — сможет ли он получить доступ и к серверу резервных копий? А наоборот?

Знание своих RPO и RTO станет определяющим при выработке вами стратегии резервного копирования. Нужна ли возможность воссоздания баз на конкретный момент времени или достаточно восстановить данные с копии, снятой прошлой ночью, смирившись с потерей всей проделанной с тех пор работы? Если восстановление на конкретный момент времени необходимо, то, вероятно, придется выполнять регулярное копирование и включить режим записи в двоичный журнал, чтобы можно было вернуть данные, а затем накатить на них журналы до нужного момента.

Вообще говоря, чем больше данных вы готовы потерять, тем проще процедура резервного копирования. Если же требования очень жесткие, то гарантировать воссоздание всей информации гораздо сложнее. Существуют даже разные варианты восстановления на конкретный момент времени. Мягкие требования допускают воссоздание данных по состоянию, довольно близкому к моменту сбоя. Жесткие требования подразумевают необходимость восстановить все зафиксированные транзакции, даже если произошло что-то ужасное, например, сервер в буквальном смысле сгорел. Для этого применяют специальные методы: хранят двоичные журналы в отдельном SAN-томе или используют DRBD-репликацию диска.

Оперативное или автономное резервное копирование?

Остановить сервер MySQL на время резервного копирования, если это допустимо, — самый безопасный и просто наилучший способ получить копию данных с минимальной вероятностью внести искажения или несогласованность. Если сервер остановлен, то информацию можно копировать, не заботясь о таких неприятностях, как «грязные» буферы в буферном пуле InnoDB или в других кэшах. Не нужно беспокоиться о том, что данные модифицируются в процессе резервного копирования, а поскольку сервер не испытывает никакой нагрузки со стороны приложения, то копирование происходит быстрее.

Однако вывод сервера из эксплуатации более затратен, чем может показаться. Даже если удается свести к минимуму само время простоев, останов и запуск MySQL могут занять довольно много времени, когда нагрузка и объем данных велики, что обсуждалось в главе 8. Мы показали некоторые методы минимизации этого воздействия, но его нельзя свести к нулю. В результате вам почти наверняка придется создавать резервные копии без останова сервера. И в зависимости от требований к согласованности создание резервной копии во время работы сервера может привести к значительным прерываниям в работе.

Одной из наибольших проблем при реализации многих методик резервного копирования является использование ими команды `FLUSH TABLES WITH READ LOCK`. Тем самым мы говорим MySQL, чтобы она закрыла и заблокировала все таблицы, сбросила данные файлов MyISAM (но не InnoDB) на диск, а также опустошила кэш запросов. На это требуется время, сколько конкретно — предсказать невозможно: оно окажется большим, если для получения глобальной блокировки чтения придется ждать завершения выполнения длительной команды или если количество таблиц велико. Пока блокировки не будут освобождены, изменить данные на сервере невозможно. Все процессы будут заблокированы и станут ожидать в очередях[1]. Выполнение

[1] Да, даже запросы SELECT окажутся заблокированными, поскольку обязательно будет запрос, который попытается изменить некоторые данные. И как только он начнет ожидать блокировки записи в таблице, все запросы, пытающиеся получить блокировки чтения, тоже должны будут ждать.

команды `FLUSH TABLES WITH READ LOCK` не так затратно, как останов сервера, потому что бóльшая часть данных остается кэшированной в памяти и сервер «прогрет», но команда все равно мешает нормальной работе. Всякий, кто говорит, что она выполняется быстро, вероятно, пытается вам что-то продать и никогда не работал на реальном сервере MySQL.

Лучший способ избежать использования `FLUSH TABLES WITH READ LOCK` — задействовать только таблицы InnoDB. Вы не можете не использовать таблицы MyISAM для привилегий и другой системной информации, но если эти данные редко меняются (что нормально), вы можете сбрасывать и блокировать только эти таблицы, не создавая проблем.

Перечислим несколько факторов, связанных с производительностью, которые следует учитывать при планировании резервного копирования.

- *Время блокировки.* Как долго вам нужно удерживать такие глобальные блокировки, как `FLUSH TABLES WITH READ LOCK`, при резервном копировании?
- *Время резервного копирования.* Сколько времени требуется, чтобы передать резервную копию в пункт назначения?
- *Резервная загрузка.* Как сильно влияет на производительность сервера передача резервной копии в пункт назначения?
- *Время восстановления.* Сколько времени занимает передача резервной копии из хранилища на сервер MySQL, повторное использование двоичных журналов и т. д.?

Самое главное — найти компромисс между временем резервного копирования и временем восстановления. Как правило, один показатель можно улучшить за счет другого, например, вы можете повысить приоритет резервного копирования ценой сокращения производительности сервера.

Можно проектировать резервное копирование так, чтобы использовались преимущества паттернов загрузки. Например, если сервер загружен только на 50 % за 8 часов ночного времени, можете попытаться спроектировать резервное копирование так, чтобы он оказался загружен менее чем на 50 % и весь процесс завершился в течение 8 часов. Это можно сделать по-разному: например использовать утилиты ionice и nice, чтобы отдать предпочтение операциям копирования или сжатия, применить разные уровни сжатия или сжимать данные на сервере резервного копирования вместо сервера MySQL. Можно также применять утилиты lzo или pigz для более быстрого сжатия. Вы можете использовать функции `O_DIRECT` или `fadvise()`, чтобы не задействовать кэш операционной системы для операций копирования и тем самым не загрязнять кэши сервера. Такие инструменты, как Percona XtraBackup и MySQL Enterprise Backup, также имеют параметры пропуска тактов, и вы можете использовать утилиту pv с параметром `--rate-limit` для ограничения пропускной способности написанных вами скриптов.

Логическое и физическое резервное копирование

Существует два основных способа резервного копирования данных в MySQL: логическое (такая копия также называется *дампом*) и путем копирования исходных файлов. В логической резервной копии данные представлены в формате, который MySQL может интерпретировать как SQL-команды или текст с разделителями[1]. Исходные файлы — это просто файлы в том виде, в каком они находятся на диске.

У каждого способа копирования данных имеются свои плюсы и минусы.

Логические резервные копии

У логических резервных копий есть следующие достоинства.

- Это обычные файлы, которые можно обрабатывать в стандартных текстовых редакторах и с помощью таких инструментов командной строки, как `grep` или `sed`. Это очень полезно, когда требуется вернуть информацию или просто просмотреть ее.
- Из них легко восстанавливать данные. Достаточно просто подать файл по конвейеру на вход программы mysql или воспользоваться программой mysqlimport.
- Резервное копирование и возврат данных можно выполнять по сети, то есть не на той же машине, где работает сервер MySQL.
- Они могут работать в таких системах, как Amazon RDS, где у вас нет доступа к базовой файловой системе.
- Процедуру можно очень гибко настраивать, потому что mysqldump — инструмент, которым многие предпочитают пользоваться для снятия логических копий, — принимает множество параметров, например раздел `WHERE`, позволяющий указать, какие строки включать в резервную копию.
- Они не зависят от подсистемы хранения. Поскольку для создания логической копии данные запрашиваются у сервера MySQL, то различия между подсистемами хранения нивелируются. Следовательно, очень просто снять копию таблицы типа InnoDB и восстановить ее в таблицу типа MyISAM. С физическими резервными копиями такой фокус не пройдет.
- Они могут помочь избежать повреждения данных. Если диски сбоят, то при копировании физических файлов получится поврежденная резервная копия, и если

[1] Логические резервные копии, создаваемые утилитой mysqldump, не всегда являются текстовыми файлами. SQL-дамп может содержать текст в различных кодировках и даже двоичные данные, которые по определению не соответствуют печатаемым символам. Да и строки могут оказаться слишком длинными для многих редакторов. И все же, как правило, такие файлы можно открыть и прочитать в текстовом редакторе, особенно если mysqldump запускалась с флагом --hex-blob.

вы специально не проверите ее по окончании резервного копирования, то узнаете об этом только тогда, когда не сможете воспользоваться ею для восстановления. Но если данные MySQL в памяти не повреждены, то иногда можно получить заслуживающую доверия логическую копию, когда снять хорошую физическую не удается

Есть у логических резервных копий и недостатки.

- Для их генерации требуется, чтобы сервер работал, так что процессор оказывается сильнее загруженным.
- В некоторых случаях логические копии оказываются объемнее исходных физических файлов[1]. ASCII-представление данных не всегда так же эффективно, как внутреннее представление в подсистеме хранения. Например, для хранения целого числа необходимо 4 байта, но для записи его в виде ASCII-текста может понадобиться до 12 символов. Зачастую логические копии хорошо поддаются сжатию, но для этого нужно дополнительное процессорное время. (Если используется много индексов, то логические резервные копии обычно меньше, чем исходные резервные копии.)
- Копирование и восстановление данных не всегда гарантирует их получение в исходном виде. Утрата точности в представлении чисел с плавающей точкой, ошибки и т. п. могут вызывать проблемы, хоть это происходит и редко.
- Для восстановления данных из логической копии необходимо загружать и интерпретировать команды и перестраивать индексы, что заставляет сервер еще больше работать.

Но главный недостаток — затраты на выгрузку данных из MySQL и обратную их загрузку с помощью команд SQL. Если вы используете логическое резервное копирование, стоит проверить время, необходимое для восстановления данных.

Инструмент mysqldump, включенный в Percona Server, может помочь, если вы работаете с таблицами InnoDB. Дело в том, что он форматирует вывод так, что при его повторной загрузке используется код быстрого создания индекса InnoDB. Проведенное нами тестирование показало, что время восстановления может сократиться на две трети или даже больше. Чем больше индексов, тем это выгоднее.

Физические резервные копии

У физических резервных копий есть следующие достоинства.

- Для получения физической копии следует просто скопировать нужные файлы в другое место. Никакой дополнительной работы для их генерации не требуется.
- Трудоемкость возврата данных из физической копии может быть меньшей и зависит от подсистемы хранения. При использовании MyISAM достаточно просто

[1] Наш опыт показывает, что обычно логические резервные копии меньше физических, но так бывает не всегда.

скопировать файлы в исходное место, тогда как для InnoDB требуется остановить сервер и, возможно, предпринять еще какие-то действия.

- Физические копии данных InnoDB и MyISAM можно переносить между платформами, операционными системами и версиями MySQL. (Логические дампы тоже. Мы просто укажем на это, чтобы исключить любые проблемы, которые могут у вас возникнуть.)
- Восстановление с физических копий может оказаться более быстрым, потому что серверу не нужно выполнять SQL-команды и выстраивать индексы. При наличии таблиц InnoDB, которые не помещаются целиком в память сервера, восстановление данных из физических файлов можно выполнить гораздо быстрее — на порядок или даже больше. На самом деле одной из самых неприятных вещей в логических резервных копиях является непредсказуемое время восстановления.

Но есть и недостатки.

- Объем физических файлов InnoDB, как правило, гораздо больше объема соответствующих логических копий. Обычно в табличном пространстве InnoDB очень много неиспользуемого места. К тому же какое-то пространство отведено под цели, не связанные с хранением табличных данных (буфер вставки, сегмент отката и т. д.).
- Не всегда физическую копию можно перенести на другую платформу, в операционную систему или версию MySQL. В частности, препятствием могут стать чувствительность к регистру букв и формат чисел с плавающей точкой. Перенос файлов в систему с другим форматом чисел с плавающей точкой вообще невозможен (впрочем, в большинстве современных процессоров применяется формат IEEE).

Работать с физическими копиями обычно проще и эффективнее[1]. Но не следует целиком полагаться на них при необходимости долговременного хранения или удовлетворения требований законодательства, время от времени нужно делать и логические копии.

Не считайте резервную копию (особенно физическую) пригодной для использования, пока не проверили ее. В случае InnoDB это означает, что нужно запустить экземпляр MySQL, дать InnoDB завершить процедуру восстановления, а затем выполнить команду `CHECK TABLES`. Можно опустить этот шаг или просто проверить файлы утилитой innochecksum, но мы советуем так не делать. Для MyISAM следует выполнить команду `CHECK TABLES` или воспользоваться утилитой myisamchk. Для всех таблиц можно использовать команду `CHECK TABLES` с утилитой mysqlcheck.

Рекомендуем комбинировать оба подхода: снять физические копии, а потом запустить экземпляр MySQL и утилиту mysqlcheck. Затем, по крайней мере время от времени, выполнить дамп данных с помощью утилиты mysqldump для получения логической резервной копии. В результате вы возьмете все лучшее от обоих подходов, не создавая на промышленном сервере излишней нагрузки по формированию

[1] Стоит отметить, что вероятность возникновения ошибок в физических резервных копиях больше — простоту mysqldump превзойти сложно.

дампа. Это особенно удобно, когда есть возможность делать мгновенные снимки файловой системы: вы создаете снимок, копируете его на другой сервер, разворачиваете, а затем проверяете физические файлы и выполняете логическое резервное копирование.

Что копировать

Требования к восстановлению диктуют, что нужно копировать. Простейшая стратегия состоит в том, чтобы скопировать данные и определения таблиц, однако это лишь необходимый минимум. Обычно для полного восстановления промышленного сервера требуется гораздо больше. Перечислим кое-что из того, что стоит включать в состав резервной копии MySQL.

- *Неочевидные данные.* Не забудьте о данных, которые не бросаются в глаза, например о двоичных журналах и журналах транзакций InnoDB.
- *Код.* В современном сервере MySQL может содержаться большой объем программного кода, в частности триггеры и хранимые процедуры. Если вы выполняете резервное копирование базы данных mysql, то большая часть этого кода в нее войдет. Однако тогда становится трудно восстановить единственную базу из всего множества, поскольку часть данных этой базы, к примеру хранимые процедуры, на самом деле содержится в базе mysql.
- *Конфигурация репликации.* Для восстановления сервера, участвующего в репликации, следует включать в резервную копию все необходимые для репликации файлы: двоичные журналы, журналы ретрансляции, индексные файлы журналов и `.info`-файлы. Как минимум, следует добавить результаты выполнения команды `SHOW MASTER STATUS` и/или `SHOW SLAVE STATUS`. Полезно также выполнить команду `FLUSH LOGS`, чтобы MySQL начала новый двоичный журнал. Восстановление на конкретный момент времени выполнить проще, когда отсчет ведется от начала, а не от середины журнала.
- *Конфигурация сервера.* Если потребуется восстановить данные после настоящей катастрофы, скажем, выстроить сервер с нуля в новом центре обработки данных после землетрясения, то вы оцените полезность хранения конфигурационных файлов сервера в составе резервной копии.
- *Отдельные файлы операционной системы.* Как и в случае с конфигурацией сервера, очень важно сохранить все внешние конфигурационные файлы, имеющие значение для работы операционной системы. На UNIX-сервере это могут быть таблицы `cron`, конфигурация пользователей и групп, административные сценарии и правила `sudo`.

Подобные рекомендации во многих случаях расцениваются как совет: «Копируйте все». Но если информации очень много, то это может оказаться слишком дорогим удовольствием, поэтому при выполнении резервного копирования стоит проявить изворотливость. Например, данные, двоичные журналы и конфигурационные файлы операционной системы и сервера можно копировать по отдельности.

Инкрементное и дифференциальное резервное копирование. При наличии большого объема данных общепринятой стратегией является регулярное инкрементное или *дифференциальное* копирование. Различия между ними могут быть слегка непонятными, поэтому давайте уточним термины: *дифференциальная резервная копия* — это резервная копия всего, что изменилось с момента создания последней полной резервной копии, тогда как *инкрементная резервная копия* содержит все, что изменилось с момента выполнения последней резервной копии любого типа.

Например, предположим, что вы делаете полную резервную копию каждое воскресенье. В понедельник создаете дифференциальную резервную копию всего, что изменилось с воскресенья. Во вторник есть два варианта: можно создать резервную копию всего, что было изменено с воскресенья (дифференциальная), или создавать резервные копии только данных, которые были изменены с момента резервного копирования в понедельник (инкрементная).

Как дифференциальные, так и инкрементные резервные копии являются частичными резервными копиями: они, как правило, не содержат полного набора данных, поскольку некоторые данные почти наверняка не изменились. Зачастую желательно выполнять частичные резервные копии для экономии накладных затрат на сервере, времени резервного копирования и резервного пространства. Однако некоторые частичные резервные копии в действительности не уменьшают эти затраты на сервере. Например, Percona XtraBackup и MySQL Enterprise Backup сканируют каждый блок данных на сервере, поэтому не слишком уменьшают накладные затраты, хотя и экономят время работы, много процессорного времени для сжатия и, конечно, дисковое пространство[1].

Вы можете использовать довольно любопытные современные методы резервного копирования, но чем сложнее решение, тем более рискованным это будет. Остерегайтесь скрытых опасностей, таких как создание множества поколений резервных копий, которые тесно связаны друг с другом, поскольку, если одно поколение содержит ошибку, оно может сделать недостоверными все остальные.

Поделимся некоторыми мыслями по этому поводу.

- Используйте инкрементные функции резервного копирования Percona XtraBackup или MySQL Enterprise Backup.
- Делайте резервную копию двоичных журналов. Можно также использовать `FLUSH LOGS`, чтобы начинать новый двоичный журнал после каждой резервной копии, а затем создавать резервные копии только новых двоичных журналов.
- Не копируйте таблицы, которые не изменялись. Некоторые подсистемы хранения, например MyISAM, записывают время последней модификации каждой таблицы. Узнать его можно, заглянув в файл на диске или воспользовавшись командой `SHOW TABLE STATUS`. При использовании InnoDB можно написать триггер,

[1] В настоящий момент разрабатывается функция истинного инкрементного резервного копирования для Percona XtraBackup. Она сможет создавать резервные копии блоков, которые были изменены, не сканируя все блоки.

который поможет отследить время последнего изменения, занося его в специальную маленькую таблицу, названную «Время последнего изменения». Но это нужно делать только для таблиц, которые изменяются редко, чтобы издержки были минимальными. Специальный скрипт резервного копирования сможет легко определить, какие таблицы были изменены.

Справочные таблицы, например содержащие названия месяцев на разных языках или сокращенные названия государств или регионов, имеет смысл поместить в отдельную базу данных, которая не копируется каждый раз.

- Не копируйте строки, которые не изменялись. Если записи в таблицу только добавляются (командой `INSERT`) — такова, например, таблица, где хранится протокол посещения страниц сайта, — то можно создать в ней столбец типа `TIMESTAMP` и включать в резервную копию только строки, добавленные с момента последнего копирования.
- Некоторую информацию не копируйте вовсе. Иногда это вполне оправданно — например, если имеется хранилище, которое создается из других данных и, строго говоря, является избыточным, то можно включать в копию лишь те элементы, которые нужны для воссоздания хранилища, а его само не копировать. Эта идея может оказаться здравой, даже если воссоздавать хранилище из исходных данных очень долго. Отказ от копирования всех данных может со временем принести многократно бóльшую экономию, чем удалось бы получить при наличии полной копии. Кроме того, можно не копировать некоторые временные значения, например таблицы, в которых хранятся данные о веб-сеансах.
- Создавайте резервные копии всего чего только можно, но позаботьтесь, чтобы они попали туда, где есть возможность устранения избыточности данных, например в файловую систему ZFS.

Недостаток инкрементного резервного копирования — повышенная сложность восстановления, увеличенный риск и большее время восстановления. Если у вас есть возможность снимать полные резервные копии, то мы рекомендуем ею воспользоваться, чтобы упростить работу.

В любом случае время от времени выполнять полное копирование необходимо — советуем делать это раз в неделю. Вряд ли стоит рассчитывать на то, что можно будет восстановиться, имея только инкрементные копии за год. Даже неделя — это трудоемко и рискованно.

Подсистемы хранения и согласованность

Наличие в MySQL различных подсистем хранения может существенно осложнить процедуру резервного копирования. Проблема заключается в том, как получить согласованную копию базы данных при наличии в ней таблиц произвольных типов.

Существуют два вида согласованности: согласованность данных и согласованность файлов.

Согласованность данных

При снятии резервной копии необходимо гарантировать, что данные в ней согласованы по времени. Например, в базе данных сайта электронной торговли счета-фактуры и платежи должны быть согласованы друг с другом. Восстановление платежа без соответствующего счета или наоборот — прямая дорога к неприятностям!

При оперативном резервном копировании (без остановки сервера) необходимо получить согласованные дубликаты всех взаимосвязанных таблиц. Это означает, что нельзя просто блокировать и копировать таблицы по одной, то есть процедура резервного копирования должна знать о структуре базы больше, чем хотелось бы. Если используется нетранзакционная подсистема хранения, то не остается ничего другого, как выполнить команду `LOCK TABLES` для всех таблиц, которые должны копироваться вместе, и освободить блокировки только после того, как процедура будет полностью завершена.

Встроенная в InnoDB технология MVCC способна в этом помочь. Вы можете начать транзакцию, скопировать группу взаимосвязанных таблиц и потом закоммитить транзакцию. (Не следует одновременно с этим использовать команду `LOCK TABLES`, если хотите получить согласованную копию, поскольку она неявно коммитит транзакцию, — детали см. в руководстве по MySQL.) Если установлен уровень изоляции `REPEATABLE READ` и не выполняются команды группы DDL, этот метод позволит получить идеально согласованный по времени снимок данных, не блокируя при этом работу сервера на время снятия копии.

Однако такой подход не защитит от неудачно спроектированной логики приложения. Предположим, что интернет-магазин вставляет запись о платеже, выполняет коммит транзакции, а затем вставляет счет-фактуру уже в другой транзакции. Процедура резервного копирования может начаться между двумя этими операциями, и тогда платеж войдет в копию, а счет-фактура — нет. Поэтому к использованию транзакций следует подходить со всей ответственностью, объединяя взаимосвязанные операции.

Получить согласованную логическую копию таблиц InnoDB позволяет также программа mysqldump, которая поддерживает флаг `--single-transaction`, делающий в точности то, что было описано ранее. Однако при этом могут возникать очень длинные транзакции, что при некоторых характеристиках рабочей нагрузки приводит к неприемлемо высоким затратам.

Согласованность файлов

Не менее важна внутренняя согласованность каждого файла. Например, в резервную копию не должен попасть файл, состояние которого отражает частичное выполнение большой команды `UPDATE`. Кроме того, необходимо, чтобы все скопированные файлы были согласованы друг с другом. Если внутренняя согласованность нарушена, то при восстановлении вас поджидают неприятные сюрпризы (скорее всего, данные будут

повреждены). А если взаимосвязанные объекты копируются в разные моменты времени, то они не будут согласованы между собой. Примерами могут служить файлы MyISAM с расширениями `.MYD` и `.MYI`. InnoDB будет журналировать ошибки или даже остановит сервер, если обнаружит несогласованность или повреждение данных.

В случае использования нетранзакционной подсистемы хранения, в частности MyISAM, единственный вариант — заблокировать и сбросить таблицы. Иными словами, нужно выполнить команды `LOCK TABLES` и `FLUSH TABLES`, чтобы сервер сохранил на диск накопившиеся в памяти изменения, или команду `FLUSH TABLES WITH READ LOCK`. По завершении сброса можно безопасно копировать физические файлы, в которых хранятся таблицы MyISAM.

В случае InnoDB гарантировать согласованность файлов на диске несколько сложнее. Даже после команды `FLUSH TABLES WITH READ LOCK` InnoDB не прекращает работу в фоновом режиме: потоки буфера вставки, журнала и записи продолжают сохранять изменения в журнал и файлы табличного пространства. Эти потоки задуманы асинхронными — именно выполнение работы в фоновых режимах позволяет InnoDB достигать высокого уровня конкурентности, — поэтому от команды `LOCK TABLES` они не зависят. Следовательно, нужно гарантировать не только внутреннюю согласованность каждого файла, но и то, что файлы журналов и табличного пространства копируются в один и тот же момент времени. Если резервная копия снимается, когда некоторый поток изменяет файл, или файлы журналов копируются не одновременно с файлами табличного пространства, то после восстановления данные могут оказаться поврежденными. Избежать такого развития событий можно несколькими способами.

- Дождаться, пока потоки вытеснения и объединения с буфером вставки завершат работу. Можно следить за тем, что выводит команда `SHOW INNODB STATUS`, и начинать копирование, когда не останется «грязных» или ожидающих операций записи буферов. Но на это может уйти много времени, а кроме того, приходится гадать на кофейной гуще и все равно из-за наличия фоновых потоков безопасность не гарантируется. Поэтому мы не рекомендуем применять такой подход.
- Делать согласованный снимок файлов данных и журналов с помощью такой системы, как LVM. Обязательно следует снимать файлы данных и журналов согласованно относительно друг друга — делать снимки по отдельности бессмысленно. Позже в этой главе мы еще будем обсуждать снимки LVM.
- Отправьте сигнал `STOP` в MySQL, сделайте резервную копию, а затем отправьте сигнал `CONT`, чтобы запустить MySQL снова. Это может показаться странной рекомендацией, но ее стоит обдумать, если единственной альтернативой является останов сервера во время резервного копирования. По крайней мере, этот метод не требует «прогрева» сервера после повторного запуска.

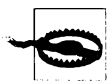
После того как файлы куда-то скопированы, можно снять блокировки и продолжить эксплуатацию сервера MySQL в обычном режиме.

Репликация

Основное преимущество резервного копирования на подчиненном сервере состоит в том, что не нужно прерывать работу и дополнительно нагружать главный. Это само по себе серьезная причина для организации подчиненного сервера, даже если он не нужен с целью балансирования нагрузки или обеспечения высокой доступности. Если бюджет ограничен, то всегда можно использовать сервер резервного копирования и в других целях, например для генерации отчетов, при условии, что вы не будете производить на нем никаких операций записи, изменяющих резервируемые данные. Подчиненный сервер не обязательно должен быть целиком выделен только для копирования, важно лишь, чтобы он успевал догнать главный к моменту снятия следующей копии, если другие занятия не позволяют ему реплицировать информацию без задержек постоянно.

Делая резервные копии на подчиненном сервере, не забывайте сохранять всю информацию о процедуре репликации, например позицию подчиненного сервера относительно главного. Это полезно для клонирования новых подчиненных серверов, повторного применения двоичных журналов к главному серверу с целью восстановления на конкретный момент времени, повышения подчиненного сервера до уровня главного и достижения многих других целей. Кроме того, убедитесь, что в момент останова подчиненного сервера нет открытых временных таблиц, поскольку это может помешать возобновлению репликации.

Для восстановления после некоторых сбоев бывает очень полезна намеренная задержка репликации. Предположим, что вы производите репликацию с паузой 1 час. Если на главном сервере была выполнена нежелательная команда, то у вас есть час, чтобы это заметить и остановить подчиненный сервер до того, как он воспроизведет это событие из своего журнала ретрансляции. Затем можно назначить подчиненный сервер главным и выполнить несколько событий из журнала, обойдя ненужные команды. Это может оказаться намного быстрее, чем применять технику восстановления на конкретный момент, которую мы обсудим в дальнейшей. Инструмент pt-slave-delay из комплекта Percona Toolkit может помочь в решении этой задачи.



Информация на подчиненном и главном серверах может различаться. Часто думают, что подчиненные серверы — точные копии главных, но наш опыт показывает, что расхождения в данных — обычное дело и у MySQL нет средств для обнаружения этой проблемы. Единственный способ выявить ее — воспользоваться таким инструментом, как pt-table-checksum из пакета Percona Toolkit

Наличие реплицированной копии может защитить от таких проблем, как расплавление диска на главном сервере, но гарантий никаких нет. Репликация — это не резервное копирование.

Управление и резервное копирование двоичных журналов

Двоичные журналы сервера — одна из важнейших вещей, которые следует включать в резервную копию. Они совершенно необходимы для восстановления на конкретный момент времени, а поскольку их размер обычно меньше, чем сами данные, то копировать их можно чаще. Если существуют резервная копия информации, снятая в какой-то момент времени, и все двоичные журналы, накопившиеся с этого момента, то эти журналы можно воспроизвести, то есть накатить изменения, произошедшие с момента создания последней полной резервной копии.

В MySQL двоичный журнал также используется для репликации. Это означает, что стратегии резервного копирования и восстановления тесно связаны с конфигурацией репликации.

Двоичные журналы — это что-то особенное. Если вы потеряли данные, то утратить еще и журналы было бы совсем некстати. Чтобы уменьшить риск потери журналов, их можно хранить в отдельном томе. Это нормально, даже если вы хотите делать снимки двоичных журналов средствами LVM. Для пущей безопасности журналы можно разместить в SAN-хранилище или реплицировать на другое устройство с помощью DRBD.

Мы рекомендуем копировать журналы почаще. Если потеря данных более чем за 30 минут недопустима, то копируйте их по крайней мере раз в 30 минут. Можно также использовать подчиненный сервер репликации, работающий в режиме только чтения, установив флаг `--log_slave_updates`. Позиции в журналах подчиненного и главного сервера, конечно, не будут совпадать, но обычно найти нужную для восстановления точку не составляет труда. Наконец, версия mysqlbinlog для MySQL 5.6 имеет очень удобную функцию для подключения к серверу и зеркалирования его двоичных журналов в реальном времени, что намного проще, чем запуск экземпляра mysqld. Он обратно совместим со старыми версиями сервера.

Наши рекомендации по конфигурации сервера для двоичного журналирования приведены в главах 8 и 10.

Формат двоичного журнала

Двоичный журнал содержит последовательность событий. У каждого события имеется заголовок фиксированной длины, в котором хранится разнообразная информация, в частности текущая временная метка и имя подразумеваемой по умолчанию базы данных. Для просмотра содержимого двоичного журнала можно воспользоваться инструментом mysqlbinlog, который распечатывает часть сведений из заголовка. Далее приведен пример вывода:

```
1  # at 277
2  #071030 10:47:21 server id
3  end_log_pos 369   Query   thread_id=13    exec_time=0
error_code=0 3  SET TIMESTAMP=1193755641/*!*/;
4  insert into test(a) values(2)/*!*/;
```

В строке 1 указано смещение в байтах от начала файла журнала (в данном случае 277).

В строке 2 приведена следующая информация:

- дата и время события, MySQL использует их для генерации команды `SET TIMESTAMP`;
- идентификатор исходного сервера, который необходим для предотвращения зацикливания репликации и других проблем;
- величина `end_log_pos`, то есть смещение начала следующего события в байтах. Для большинства событий в транзакции из нескольких команд это значение некорректно. Во время выполнения транзакции MySQL копирует события в буфер на главном сервере, но позиция следующего события в журнале в этот момент неизвестна;
- тип события. В приведенном примере это `Query`, но существует много других типов;
- идентификатор потока, обработавшего событие на исходном сервере. Он важен для аудита и для выполнения функции `CONNECTION_ID()`;
- величина `exec_time`, которая представляет собой разницу между временной меткой команды и временем, в которое команда была записана в двоичный журнал. Не стоит ориентироваться на этот показатель, поскольку его значение может быть ошибочным, если подчиненный сервер сильно отстал от главного;
- код ошибки, возникшей при обработке события на исходном сервере. Если при воспроизведении на подчиненном сервере возникнет иная ошибка, репликация на всякий случай прекратится.

В остальных строках печатаются SQL-команды, необходимые для воспроизведения события. Здесь же вы найдете переменные, определенные пользователем, и прочие специальные установки, например временную метку на момент начала обработки команды.

При использовании построчного журналирования, появившегося в версии MySQL 5.1, событие не содержит текста SQL-команд, а является образом модификаций, произведенных данной командой в таблице. Эти события не предназначены для чтения человеком.

Безопасное удаление старых двоичных журналов

Необходимо продумать стратегию удаления двоичных журналов, чтобы MySQL не заполнила ими весь диск. До какого размера вырастет журнал, зависит от его формата и рабочей нагрузки (при построчном журналировании, появившемся в версии MySQL 5.1, размер одной записи больше). Мы рекомендуем по возможности хранить журналы до тех пор, пока они не станут бесполезными. Двоичные журналы могут

пригодиться для настройки подчиненных серверов репликации, анализа рабочей нагрузки, аудита и восстановления на конкретный момент времени с помощью полной резервной копии. Учитывайте все это, когда будете решать, как долго хранить журналы.

Обычно используют конфигурационную переменную `expire_logs_days`, которая сообщает MySQL, по истечении какого времени следует удалять журналы. До выхода версии MySQL 4.1 этой переменной не существовало, поэтому журналы приходилось удалять вручную. Еще с тех времен осталась рекомендация уничтожать старые двоичные журналы такой командой в списке заданий `cron`:

```
0 0 * * * /usr/bin/find /var/log/mysql -mtime +N -name "mysql-bin.[0-9]*" | xargs rm
```

Хотя до версии MySQL 4.1 это был единственный способ удалить двоичные журналы, в более поздних версиях так делать ни в коем случае не следует! Если стереть журналы командой `rm`, то индексный файл `mysql-bin.index` рассинхронизируется с файлами на диске и некоторые команды, например `SHOW MASTER LOGS`, начнут работать неправильно, но что самое опасное — пользователь не получит никаких сообщений об ошибках. Ручная корректировка файла `mysql-bin.index` не поможет. Поэтому вставляйте в таблицу `cron` такую команду:

```
0 0 * * * /usr/bin/mysql -e "PURGE MASTER LOGS BEFORE CURRENT_DATE - INTERVAL N DAY"
```

Значение параметра `expire_logs_days` считывается на этапе запуска сервера или в момент, когда MySQL сменяет двоичный журнал, поэтому, если он никогда не заполняется до конца и не сменяется, сервер не будет удалять старые записи. Решение о том, какие файлы уничтожить, сервер принимает, исходя из даты модификации файла, а не из его содержимого.

Резервное копирование данных

Как обычно, существуют хорошие и плохие способы резервного копирования, причем самые очевидные из них не обязательно лучшие. Хитрость состоит в том, чтобы по максимуму задействовать мощность сети, диска и процессора и постараться закончить копирование как можно быстрее. Для того чтобы найти золотую середину, придется поэкспериментировать.

Снятие логической резервной копии

Нужно четко понимать, что существует две разновидности логических копий: SQL-дампы и файлы с разделителями.

SQL-дампы

SQL-дамп лучше знаком пользователям, так как это именно то, что по умолчанию создает утилита mysqldump. Например, при выгрузке дампа небольшой таблицы с параметрами по умолчанию получится такой результат (мы кое-что опустили):

```
$ mysqldump test t1
-- [Комментарий: номер версии и информация о сервере]
/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
-- [Дополнительные комментарии, зависящие от версии,
--  с параметрами, необходимыми для восстановления]
--
-- Структура таблицы `t1`
-- DROP TABLE IF EXISTS `t1`;
CREATE TABLE `t1` (
  `a` int(11) NOT NULL,
  PRIMARY KEY  (`a`)
  ) ENGINE=MyISAM DEFAULT CHARSET=latin1;
--
-- Дамп данных таблицы `t1`
-- LOCK TABLES `t1` WRITE;
/*!40000 ALTER TABLE `t1` DISABLE KEYS */;
INSERT INTO `t1` VALUES (1);
/*!40000 ALTER TABLE `t1` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;
/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
-- [Другие параметры восстановления]
```

Файл дампа содержит как описание таблицы, так и данные, причем и то и другое представлено в виде корректных SQL-команд. Файл начинается с комментариев, в которых устанавливаются значения различных параметров MySQL. Они включены для того, чтобы помочь вам при восстановлении, а также ради совместимости и корректности. Затем идут описание таблицы и данные. В самом конце сценарий воссоздает параметры, которые были изменены в начале дампа.

Созданный дамп может быть выполнен для восстановления данных. Это удобно, но подразумеваемые по умолчанию параметры mysqldump не годятся для снятия очень больших резервных копий (в дальнейшем рассмотрим параметры mysqldump подробнее).

Утилита mysqldump — не единственный инструмент, позволяющий создать логическую резервную копию. Это можно сделать, к примеру, с помощью phpMyAdmin[1]. Но мы хотим отметить не столько проблемы, свойственные тому или иному инструменту, сколько недостатки монолитной логической копии как таковой. Перечислим их.

- *Схема и данные хранятся вместе.* Хотя это удобно, когда нужно восстановить все из одного файла, но усложняет задачу, если требуется восстановить только одну таблицу или только данные. Эту трудность можно устранить, выгрузив дамп дважды: один раз для данных, другой — для схемы, но от следующих проблем все равно никуда не деться.
- *Огромные SQL-команды.* Разбор и выполнение всех SQL-команд в дампе требует от сервера значительных усилий. Поэтому загрузка данных происходит довольно медленно.
- *Один гигантский файл.* Большинство текстовых редакторов не умеют обрабатывать большие файлы или файлы с очень длинными строками. Хотя в некоторых

[1] Пожалуйста, не используйте утилиты mk-parallel-dump и mk-parallel-restore из пакета Maatkit. Они небезопасны.

случаях для извлечения нужных данных можно воспользоваться потоковыми редакторами, например sed или grep, лучше, чтобы файлы были поменьше.

- *Создание логической копии — весьма затратная процедура.* Существуют более эффективные способы извлечь из MySQL данные, помимо передачи их по протоколу взаимодействия между клиентом и сервером в виде результирующих наборов.

Все эти ограничения означают, что по мере роста таблиц SQL-дамп становится непригодным для работы. Однако существует и другая возможность: экспортировать информацию в файлы с разделителями.

Резервные копии в виде файлов с разделителями

Для создания логической копии данных в формате файла с разделителями можно воспользоваться командой `SELECT INTO OUTFILE SQL`. (Эту же команду выполнит утилита mysqldump, запущенная с флагом `--tab`.) Файлы с разделителями содержат значения, представленные в кодировке ASCII, без SQL-команд, комментариев и имен столбцов. Следующая команда выводит результат в формате с разделителями-запятыми (CSV), общепринятом для представления табличных данных.

```
mysql> SELECT * INTO OUTFILE '/tmp/t1.txt'
    -> FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    -> LINES TERMINATED BY '\n'
    -> FROM test.t1;
```

Результирующий файл компактнее, и им проще манипулировать с помощью утилит командной строки, чем SQL-дампа. Но главное достоинство кроется не в этом, а в скорости резервного копирования и последующего возврата данных. Для загрузки информации назад в таблицу предназначена команда `LOAD DATA INFILE`, которой следует указать те же параметры, что и при выгрузке:

```
mysql> LOAD DATA INFILE '/tmp/t1.txt'
    -> INTO TABLE test.t1
    -> FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    -> LINES TERMINATED BY '\n';
```

Далее для демонстрации различий в скорости создания SQL-дампа и файла с разделителями приведены результаты неофициального теста. Мы воспользовались слегка адаптированными данными из промышленной базы. Выгружаемая таблица имеет следующую структуру:

```
CREATE TABLE load_test (
    col1 date NOT NULL,
    col2 int NOT NULL,
    col3 smallint unsigned NOT NULL,
    col4 mediumint NOT NULL,
    col5 mediumint NOT NULL,
    col6 mediumint NOT NULL,
    col7 decimal(3,1) default NULL,
    col8 varchar(10) NOT NULL default '',
    col9 int NOT NULL,
    PRIMARY KEY  (col1, col2)
 ) ENGINE=InnoDB;
```

Она содержит 15 миллионов строк и занимает примерно 700 Мбайт на диске. В табл. 15.1 приведены результаты сравнения двух методов резервного копирования и восстановления. Как видите, разница в скорости восстановления весьма значительна.

Таблица 15.1. Время резервного копирования и восстановления для SQL-дампа и файла с разделителями

Метод	Размер файла, Мбайт	Время выгрузки, с	Время восстановления, с
SQL-дамп	727	102	600
Файл с разделителями	669	86	301

Однако у метода `SELECT INTO OUTFILE` есть ограничения.

- Копировать данные можно только в файл, находящийся на том же компьютере, где работает MySQL. (Но можно написать собственный вариант команды `SELECT INTO OUTFILE` в виде программы, которая читает результат `SELECT` и записывает его на диск. Мы знаем людей, которые так и делали.)
- У MySQL должно быть разрешение на запись в каталог, где создается файл, поскольку именно сервер MySQL, а не пользователь, запустивший команду, пишет в этот файл.
- Из соображений безопасности MySQL отказывается перезаписывать существующий файл вне зависимости от имеющихся разрешений.
- Невозможно записывать дамп в сжатый файл.
- С экспортом и импортом некоторых элементов, например нестандартных наборов символов, могут быть проблемы.

Снимки файловой системы

Снимок файловой системы — прекрасный способ сделать оперативную резервную копию. Файловые системы, поддерживающие снимки, способны мгновенно создать согласованный образ содержимого, который затем можно использовать в качестве резервной копии. К числу таких файловых систем и устройств относятся FreeBSD, ZFS, GNU/Linux Logical Volume Manager (LVM), а также многие SAN-системы и файловые хранилища, например NetApp.

Не путайте снимок с резервной копией. Снимок лишь позволяет уменьшить время удержания блокировок, после того как блокировки сняты, файлы необходимо поместить в резервную копию. На самом деле можно даже делать снимки таблиц InnoDB без захвата блокировок. Мы покажем два способа использования LVM для резервного копирования базы, состоящей только из таблиц InnoDB: с минимальной блокировкой и вообще без блокировки.

Снимки могут быть отличным способом сделать резервную копию для определенных целей. Например, в случае возникновения проблемы во время обновления.

Вы можете сделать снимок, обновиться, а в случае возникновения проблемы просто откатиться назад к снимку. То же самое можно выполнить для любой операции, которая является неопределенной и рискованной, например для изменения огромной таблицы, которое занимает неопределенное время.

Как работают снимки LVM

В LVM для создания снимка применяется технология копирования при записи, а это означает, что логическая копия всего тома снимается мгновенно. Чем-то это напоминает технологию MVCC в базах данных, отличие заключается в том, что хранится лишь одна версия устаревшей информации.

Обратите внимание: мы не говорим о физической копии. Может показаться, что логическая копия содержит данные, которые были в томе в момент снимка, но на самом деле изначально в ней нет вообще никаких данных. Вместо того чтобы копировать информацию в снимок, LVM лишь запоминает момент времени, в который он был создан. Позже, когда вы будете запрашивать данные из снимка, LVM начнет читать информацию из ее исходного местоположения. Поэтому-то операция первоначального копирования производится мгновенно независимо от размера тома, для которого создается снимок.

Когда что-то меняется в данных в исходном томе, LVM копирует прежнее содержимое блоков в область, зарезервированную для снимка, и лишь затем вносит модификации. LVM не хранит несколько старых версий данных, поэтому последующая запись в уже измененные блоки не требует никакой дополнительной работы. Иными словами, только при первой записи блок копируется в зарезервированную область.

Когда кто-то запрашивает эти блоки из снимка, LVM читает данные из копий в зарезервированной области, а не с рабочего тома. Это позволяет неизменно видеть в снимке одну и ту же информацию, хотя блокировки тома ни в какой момент не производилось. Эта схема изображена на рис. 15.1.

Рис. 15.1. Технология копирования при записи сокращает место, необходимое для хранения снимка тома

Снимку соответствует новое логическое устройство в каталоге /dev, которое можно смонтировать, как и любое другое.

Теоретически можно сделать снимок огромного тома, затратив на это очень немного физического пространства. Однако необходимо заранее оценить, сколько блоков может обновиться, пока снимок открыт, и отвести для них достаточно места. Если для копирования при записи не хватит дискового пространства, то снимок переполнится и устройство станет недоступно. Выглядит это как отсоединение внешнего диска: любая операция чтения с устройства, в том числе резервное копирование, завершается с ошибкой ввода/вывода.

Необходимые условия и конфигурация

Создание снимка — почти тривиальная задача, но нужно сконфигурировать систему так, чтобы получать согласованный дубликат всех файлов, которые планируется включить в резервную копию, в один и тот же момент времени. Прежде всего убедитесь, что система удовлетворяет следующим условиям.

- Все файлы InnoDB (файлы табличного пространства и журналы транзакций InnoDB) должны находиться в одном физическом томе (разделе). Необходима абсолютная согласованность во времени, а LVM не умеет одномоментно делать согласованные снимки нескольких томов. (Это ограничение LVM, в некоторых других системах таких проблем нет.)
- Если необходимо включить в резервную копию также определения таблиц, то в том же томе должен находиться каталог данных MySQL. Если для резервного копирования определений таблиц применяется какой-то другой метод, например хранение схемы в системе управления версиями, то об этом можно не беспокоиться.
- В группе томов должно быть достаточно свободного места для создания снимка, сколько именно — зависит от рабочей нагрузки. Поэтому при конфигурировании системы оставьте нераспределенное пространство для последующих снимков.

В LVM существует понятие *группы томов*, в которую входит один или несколько логических томов. Посмотреть, какие имеются группы томов, можно следующим образом:

```
# vgs
  VG   #PV #LV #SN Attr   VSize   VFree
  vg     1   4   0 wz--n- 534.18G 249.18G
```

Здесь мы видим, что существует одна группа, состоящая из четырех логических томов, которые находятся в одном физическом томе, в котором свободно примерно 250 Гбайт. При необходимости можно получить более подробную информацию с помощью утилиты vgdisplay. Теперь посмотрим на сами логические тома:

```
# lvs
  LV    VG   Attr   LSize   Origin Snap%  Move Log Copy%
  home  vg   -wi-ao  40.00G
  mysql vg   -wi-ao 225.00G
  tmp   vg   -wi-ao  10.00G
  var   vg   -wi-ao  10.00G
```

Мы видим, что размер тома mysql составляет 225 Гбайт. Данное устройство называется /dev/vg/mysql. Это всего лишь имя, хотя выглядит оно как путь в файловой системе. Еще сильнее запутывает ситуацию то, что существует символическая ссылка с этого имени на узел реального устройства /dev/mapper/vg-mysql, которую показывают команды ls и mount:

```
# ls -l /dev/vg/mysql
lrwxrwxrwx 1 root root 20 Sep 19 13:08 /dev/vg/mysql -> /dev/mapper/vg-mysql
# mount | grep mysql
 /dev/mapper/vg-mysql on /var/lib/mysql
```

Вооружившись этой информацией, вы можете приступить к созданию снимка файловой системы.

Создание, монтирование и удаление снимка LVM

Создать снимок можно одной командой. Нужно лишь решить, куда его поместить и сколько места отвести под копирование при записи. Не бойтесь указать больше места, чем может реально потребоваться. LVM не займет этот объем немедленно, а просто зарезервирует для использования в будущем, поэтому резервирование очень большого пространства не принесет никакого вреда, если только вам одновременно не нужно оставлять место для других снимков.

Давайте попрактикуемся в создании снимка. Отведем для копирования при записи 16 Гбайт и назовем снимок backup_mysql:

```
# lvcreate --size 16G --snapshot --name backup_mysql /dev/vg/mysql
Logical volume "backup_mysql" created
```

Мы сознательно назвали том backup_mysql, а не mysql_backup, чтобы не возникало неоднозначности при автоматическом завершении команды после нажатия клавиши Tab. Это позволяет подстраховаться от случайного удаления тома mysql при автодополнении имени файла нажатием кнопки Tab.

Теперь посмотрим на состояние только что созданных томов:

```
# lvs
  LV          VG   Attr   LSize   Origin Snap%  Move Log Copy%
  backup_mysql vg   swi-a-  16.00G mysql    0.01
  home         vg   -wi-ao  40.00G
  mysql        vg   owi-ao 225.00G
  tmp          vg   -wi-ao  10.00G
  var          vg   -wi-ao  10.00G
```

Обратите внимание на то, что атрибуты снимка и исходного устройства различаются и что показана кое-какая дополнительная информация: источник снимка и сведения о том, какая часть из отведенных 16 Гбайт в данный момент уже израсходована на копирование при записи. Мы настоятельно рекомендуем следить за состоянием во

время снятия резервной копии, чтобы не пропустить момент, когда снимок близок к заполнению и грозит отказом. Вести мониторинг состояния устройств можно интерактивно или с помощью таких систем, как Nagios.

```
# watch 'lvs | grep backup'
```

Как видно из представленной ранее выдачи команды `mount`, том `mysql` содержит файловую систему. Следовательно, она же будет и в томе снимка и ее можно монтировать и использовать, как и любую другую файловую систему:

```
# mkdir /tmp/backup
# mount /dev/mapper/vg-backup_mysql /tmp/backup
# ls -l /tmp/backup/mysql
total 5336
-rw-r----- 1 mysql mysql       0 Nov 17  2006 columns_priv.MYD
-rw-r----- 1 mysql mysql    1024 Mar 24  2007 columns_priv.MYI
-rw-r----- 1 mysql mysql    8820 Mar 24  2007 columns_priv.frm
-rw-r----- 1 mysql mysql   10512 Jul 12 10:26 db.MYD
-rw-r----- 1 mysql mysql    4096 Jul 12 10:29 db.MYI
-rw-r----- 1 mysql mysql    9494 Mar 24  2007 db.frm
... опущено ...
```

Поскольку мы лишь практикуемся, размонтируем и удалим снимок командой `lvremove`:

```
# umount /tmp/backup
# rmdir /tmp/backup
# lvremove --force /dev/vg/backup_mysql
  Logical volume "backup_mysql" successfully removed
```

Применение снимков LVM для оперативного резервного копирования

Мы научились создавать, монтировать и удалять снимки, теперь воспользуемся ими для снятия резервных копий. Сначала посмотрим, как сделать резервную копию базы данных с таблицами InnoDB, не останавливая сервер MySQL. Подключитесь к серверу и сбросьте таблицы на диск, установив глобальную блокировку чтения, а затем получите координаты в двоичном журнале:

```
mysql> FLUSH TABLES WITH READ LOCK; SHOW MASTER STATUS;
```

Запишите результаты, выданные командой `SHOW MASTER STATUS`, и оставьте соединение с MySQL открытым, чтобы не освобождать блокировку. Теперь можно сделать снимок LVM и сразу же снять блокировку чтения, выполнив команду `UNLOCK TABLES` либо закрыв соединение. И наконец, смонтируйте снимок и поместите файлы в резервную копию.

Основная проблема при таком подходе заключается в том, что на получение блокировки чтения может уйти довольно много времени, особенно если выполняется длинный запрос. Пока соединение ожидает глобальной блокировки чтения, все запросы также блокируются и невозможно заранее сказать, как долго это будет продолжаться.

Снимки файловой системы и InnoDB

Фоновые потоки InnoDB продолжают работать даже после блокировки всех таблиц, поэтому они могут вести запись в файлы, когда делается снимок. Кроме того, поскольку подсистема InnoDB не выполняла корректную последовательность останова, файлы в снимке будут выглядеть так, будто сервер был неожиданно отсоединен от сети.

Это не проблема, поскольку InnoDB — система ACID. В любой момент, в том числе в момент снятия снимка, любая закоммиченная транзакция находится либо в файлах данных InnoDB, либо в файлах журналов. После того как MySQL будет запущена на восстановленном снимке, InnoDB инициирует свою процедуру восстановления точно так же, как в случае отключения питания сервера. Она найдет в журнале все зафиксированные транзакции, которые еще не попали в файлы данных, и применит их, поэтому потери транзакций не произойдет. Вот почему так важно включать файлы данных и журналов InnoDB в один снимок.

По той же причине следует тестировать резервные копии сразу после снятия. Запустите экземпляр MySQL, сообщите ему, где находится новая копия, позвольте InnoDB завершить восстановление и проверьте все таблицы. Тогда вы будете уверены, что в резервную копию незаметно для вас не просочились поврежденные данные (файлы могут оказаться испорченными по самым разным причинам). У такого подхода есть и еще один плюс: в будущем воссоздание информации с этой копии пройдет быстрее, так как вы уже запустили процесс восстановления.

Описанные действия можно проделать над снимком еще до того, как он будет помещен в резервную копию, хотя это добавляет затрат. Но обязательно запланируйте его, подробнее об этом — позже.

Применение снимков LVM для резервного копирования InnoDB без блокировок

Техника резервного копирования без блокировок отличается от описанной незначительно. Различие в том, что не выполняется команда `FLUSH TABLES WITH READ LOCK`. Это означает, что для файлов MyISAM на диске нет никаких гарантий согласованности, но если все таблицы имеют тип InnoDB, это не имеет существенного значения. В системной базе данных `mysql` все равно есть кое-какие таблицы типа MyISAM, но в условиях типичной рабочей нагрузки маловероятно, что они будут изменяться в момент снятия снимка.

Если вы полагаете, что системные таблицы в базе `mysql` все же могут изменяться, то заблокируйте их и сбросьте на диск. К таким таблицам не предъявляются длительные запросы, поэтому обычно все происходит очень быстро:

```
mysql> LOCK TABLES mysql.user READ, mysql.db READ, ...;
mysql> FLUSH TABLES mysql.user, mysql.db, ...;
```

Поскольку глобальная блокировка на чтение не ставилась, то ничего полезного от команды `SHOW MASTER STATUS` не получить. Однако при запуске MySQL на снимке для проверки целостности копии вы увидите в журнале примерно такие сообщения:

```
InnoDB: Doing recovery: scanned up to log sequence number 0 40817239
InnoDB: Starting an apply batch of log records to the database...
InnoDB: Progress in percents: 3 4 5 6 ...[omitted]... 97 98 99
InnoDB: Apply batch completed
InnoDB: Last MySQL binlog file position 0 3304937, file name /var/log/mysql/
mysql-bin.000001 070928 14:08:42
InnoDB: Started; log sequence number 0 40817239
```

InnoDB журналирует в двоичном журнале позицию, соответствующую точке, до которой он был восстановлен. Именно эту позицию можно использовать при восстановлении на конкретный момент времени.

У описанного подхода к резервному копированию без блокировки с помощью снимков в версии MySQL 5.0 и более поздних есть один недостаток. В этих версиях для координации между InnoDB и двоичным журналом применяется технология XA (распределенные транзакции). Если восстановить копию на сервер с идентификатором `server_id`, отличным от того, на котором снималась копия, то сервер может обнаружить транзакции, подготовленные сервером с идентификатором, не совпадающим с его собственным. В таком случае сервер рискует запутаться, и после восстановления транзакция может зависнуть в состоянии `PREPARED`. Хоть и редко, но такое случается. Поэтому обязательно проверяйте копию — не считайте, что с ней априори все нормально. Может оказаться, что восстановиться с нее не удастся!

Если снимок выполнялся на подчиненном сервере, то процедура восстановления InnoDB напечатает также такие строки:

```
InnoDB: In a MySQL replica the last master binlog file
InnoDB: position 0 115, file name mysql-bin.001717
```

Эти строчки содержат координаты в двоичном журнале главного (а не подчиненного) сервера той точки, до которой прошло восстановление. Это может оказаться очень полезным при создании резервных копий на подчиненных серверах или для клонирования одних подчиненных серверов из других.

Планирование резервного копирования с помощью LVM

Резервное копирование с помощью снимков LVM не дармовое удовольствие. Чем больше сервер записывает в исходный том, тем больше издержки. Когда сервер модифицирует много блоков, расположенных в случайном порядке, головка диска должна постоянно перемещаться в зарезервированную для копирования область, чтобы записать туда старую версию данных. Чтение из снимка также сопряжено с издержками, так как бóльшую часть информации LVM читает из исходного тома, а из зарезервированной области — только по необходимости. Таким образом, логически последовательное чтение из снимка на самом деле приводит к хаотичному перемещению головки.

Это необходимо учитывать при планировании. По существу, получается, что как при чтении, так и при записи исходный том и снимок функционируют хуже, чем обычно, а если в зарезервированной области оказалось много блоков, то, вероятно, даже гораздо хуже. В результате может замедлиться как работа сервера MySQL, так и процесс копирования файлов в резервную копию. Мы провели эталонное тестирование и выяснили, что затраты на снимок LVM намного больше, чем следовало бы, — мы обнаружили, что производительность может оказаться в пять раз ниже в зависимости от рабочей нагрузки и файловой системы. Помните об этом, планируя резервное копирование.

Кроме того, важно выделить достаточно места для снимка. Мы рекомендуем следующую тактику.

- Помните, что LVM должен скопировать каждый измененный блок в снимок только один раз. Когда MySQL записывает блок в исходный том, LVM копирует блок в снимок, а затем вносит пометку о скопированном блоке в таблицу исключений. При последующих изменениях того же блока он уже не копируется.
- Если в базе имеются только таблицы типа InnoDB, то примите во внимание, как InnoDB сохраняет информацию. Поскольку любой элемент данных записывается дважды, то по меньшей мере половина всех операций ввода/вывода попадает в буфер двойной записи, файлы журналов и сравнительно компактные области на диске. Тем самым одни и те же блоки перезаписываются снова и снова, так что при первой модификации они оказывают влияние на снимок, но потом сохранение туда прекращается.
- Затем оцените, какая часть операций ввода/вывода сопряжена с записью в блоки, которые еще не копировались в снимок, по сравнению с повторной модификацией уже измененных данных. Не бойтесь завысить оценку.
- С помощью vmstat или iostat соберите статистику о количестве блоков, записываемых в секунду.
- Измерьте (или оцените), сколько времени требуется для копирования резервной копии в другое место, иными словами, как долго нужно будет держать снимок LVM открытым.

Предположим, вы прикинули, что половина операций записи пойдет в область, зарезервированную для копирования при записи, и что сервер сохраняет 10 Мбайт в секунду. Если для копирования снимка на другой сервер требуется 1 час (3600 секунд), то для снимка понадобится 1/2 × 10 Мбайт × 3600 = 18 Гбайт. На всякий случай оставьте побольше.

Иногда бывает легко вычислить, сколько данных изменится, пока снимок открыт. Вернемся к примеру, который встречался уже неоднократно. Система поиска по форумам BoardReader хранит на каждом узле примерно 1 Тбайт данных в виде таблиц InnoDB. Однако мы знаем, что наибольшие затраты обусловлены загрузкой новых данных. Каждый день добавляется примерно 10 Гбайт информации, поэтому для снимка вполне хватит 50 Гбайт. Впрочем, эта оценка не всегда корректна. Как-то раз мы запустили длительную команду **ALTER TABLE**, которая последовательно изменяла

каждый шард, и в сумме было модифицировано гораздо больше чем 50 Гбайт — пока эта команда работала, мы не могли снять резервную копию. Чтобы избежать таких проблем, стоит после создания моментального снимка некоторое время подождать, поскольку пик дополнительной загрузки наблюдается сразу после его создания.

Миф о резервном копировании № 2: «Мой снимок — это моя резервная копия»

Снимок, будь то снимок LVM, моментальный снимок ZFS или моментальный снимок SAN, не является реальной резервной копией, поскольку не содержит полной копии ваших данных. Поскольку копии являются копиями при записи, они содержат только различия между фактической копией данных и данными в тот момент, когда был сделан снимок. Если немодифицированный блок оказывается поврежденным в фактической копии данных, то хорошая копия этого блока, которую можно было бы использовать для восстановления, отсутствует и каждый снимок видит тот же поврежденный блок, что и существующий том. Используйте снимки, чтобы «зафиксировать» свои данные во время резервного копирования, но не полагайтесь на сам снимок как на резервную копию.

Другие варианты применения и альтернативы

Снимки можно использовать не только для резервного копирования. Например, они полезны для записи контрольной точки непосредственно перед выполнением опасной операции. Некоторые файловые системы позволяют вернуть данные из снимка на основную файловую систему. Это позволяет без труда откатиться к состоянию перед созданием снимка.

Снимки файловой системы не единственный способ получить мгновенную копию данных. Еще один вариант — это расщепление RAID-массива: если имеется зеркалированный программно реализованный RAID-массив из трех дисков, то можно исключить один диск из зеркала и смонтировать его отдельно. Тогда никаких затрат на копирование при записи не будет, а в случае необходимости очень легко сделать такой снимок главной копией. Однако после добавления этого диска обратно в RAID-массив его придется повторно синхронизировать. К сожалению, бесплатного сыра не бывает.

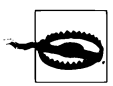
Восстановление из резервной копии

Порядок восстановления зависит от того, как снималась резервная копия. Вам может понадобиться выполнить все или некоторые из перечисленных далее шагов.

- Остановить сервер MySQL.
- Записать куда-нибудь конфигурацию сервера и права доступа к файлам.
- Скопировать данные с резервной копии в каталог данных MySQL.

- Внести изменения в конфигурационные файлы.
- Изменить права доступа к файлам.
- Запустить сервер в режиме ограниченного доступа и подождать, пока он перейдет в состояние готовности.
- Загрузить логические файлы резервной копии.
- Проверить и воспроизвести двоичные журналы.
- Убедиться, что все восстановлено.
- Перезапустить сервер в режиме полного доступа.

В следующих разделах мы покажем, как выполняется каждый из этих шагов. А позже сделаем некоторые замечания о конкретных методах резервного копирования и соответствующих инструментах.



Если можно предположить, что текущие версии файлов еще понадобятся, не замещайте их файлами из резервной копии. Например, если копия не содержит двоичных журналов, а они необходимы для восстановления на конкретный момент времени, то не затирайте существующие журналы устаревшими версиями из копии. При необходимости переименуйте их или скопируйте в другое место.

Во время восстановления часто бывает важно, чтобы к MySQL не мог обращаться никто, кроме процесса восстановления. Чтобы гарантировать, что сервер будет недоступен для приложений, пока мы все не проверим, запускаем его с флагами `--skip-networking` и `--socket=/tmp/mysql_recover.sock`. Это особенно важно для логических копий, которые загружаются по частям.

Восстановление из физических файлов

Процедура возврата данных из физических файлов довольно прямолинейна, иначе говоря, вариантов здесь немного. Хорошо это или плохо, зависит от требований к восстановлению. Обычно все сводится к простому копированию файлов туда, где им надлежит находиться.

Нужно ли останавливать MySQL, определяет подсистема хранения. Файловые объекты MyISAM обычно не зависят друг от друга, поэтому простого копирования файлов с расширениями `.frm`, `.MYI` и `.MYD` для каждой таблицы достаточно, даже на работающем сервере. Сервер найдет таблицу, как только к ней поступит запрос или будет выполнена иная команда, для которой необходима данная таблица (например, `SHOW TABLES`). Если во время копирования этих файлов таблица была открыта, то вполне возможны неприятности, поэтому предварительно удалите или переименуйте ее либо заблокируйте командами `LOCK TABLES` и `FLUSH TABLES`.

С InnoDB дело обстоит иначе. Если восстанавливается традиционно сконфигурированная база InnoDB, когда все таблицы хранятся в одном табличном пространстве, то нужно будет остановить MySQL, скопировать или переместить файлы на место, а затем перезапустить сервер. Кроме того, следует убедиться, что файл журнала транзакций соответствует файлам, содержащим табличное пространство. Если эти файлы не соответствуют друг другу, например, табличное пространство вы заменили, а о журнале транзакций забыли, то InnoDB откажется запускаться. Поэтому так важно включать в резервную копию не только файлы данных, но и журнал транзакций.

Если используется возможность размещать по одной таблице в каждом файле (режим `innodb_file_per_table`), то InnoDB хранит данные и индексы для каждой таблицы в файле с расширением `.ibd`, представляющем собой нечто вроде комбинации `.MYI`- и `.MYD`-файлов в подсистеме MyISAM. Резервное копирование и восстановление отдельных таблиц в этом случае может выполняться простым копированием этих файлов, и делать это можно на работающем сервере, но не так просто, как в случае MyISAM. Отдельные файлы не являются независимыми от InnoDB в целом. В каждом `.ibd`-файле записана информация, сообщающая InnoDB о том, как этот файл связан с основным (общим) табличным пространством. При возврате такого файла необходимо попросить InnoDB импортировать его.

У этой процедуры много ограничений, о которых можно прочитать в разделе руководства по MySQL, посвященном табличным пространствам с раздельным хранением таблиц. Самое серьезное заключается в том, что восстановить таблицу можно только на тот сервер, с которого она была скопирована. Не то чтобы резервное копирование и восстановление таблиц в этой конфигурации было вообще невозможно, но это несколько сложнее, чем может показаться.

В Percona Server и Percona XtraBackup сделаны усовершенствования, которые снимают некоторые ограничения этого процесса, такие как восстановление на том же сервере.

Наличие таких сложностей означает, что восстановление физических файлов способно оказаться очень трудоемким процессом, в ходе которого можно наделать ошибок. Эмпирическое правило состоит в том, что чем сложнее и труднее становится процедура восстановления, тем важнее оградить себя от неприятностей, создавая также логические копии. Всегда полезно иметь такую копию на случай, если что-то пойдет не так и вы не сможете заставить MySQL использовать физическую копию.

Запуск MySQL после восстановления физических файлов. Перед тем как запускать восстановленный сервер MySQL, нужно сделать несколько шагов.

Первое и самое важное, о чем легче всего забыть, — проверить конфигурацию сервера и убедиться в том, что для восстановленных файлов заданы правильный владелец и права доступа. Это нужно делать до попытки запустить сервер. Если эти атрибуты хоть чем-то отличаются от необходимых, MySQL может не запуститься. В различных системах они задаются по-разному, поэтому посмотрите в своих записях, как они должны быть установлены. Обычно для файлов и каталогов указываются владелец и группа `mysql`, причем владельцу и группе должны быть разрешены чтение и запись, а всем остальным запрещен всякий доступ.

Советуем также следить за журналом ошибок MySQL в ходе запуска. В UNIX-подобных системах для этого можно выполнить такую команду:

```
$ tail -f /var/log/mysql/mysql.err
```

Точное местоположение журнала ошибок зависит от системы. Начав мониторинг этого файла, запускайте сервер MySQL и наблюдайте за тем, что будет появляться в журнале. Если ошибок не возникнет, значит, сервер восстановился нормально и может работать.

Наблюдение за журналом ошибок особенно важно в последних версиях MySQL. В прежних реализациях сервер просто не запускался, если InnoDB обнаруживала ошибку, но теперь он запускается, однако InnoDB отключается. Даже если кажется, что сервер стартовал без проблем, следует в каждой базе данных выполнить команду `SHOW TABLE STATUS` и снова проверить журнал ошибок.

Восстановление из логической копии

Если восстановление производится из логической резервной копии, а не из физических файлов, то для загрузки данных в таблицы понадобится сам сервер MySQL. Копирования файлов на уровне операционной системы недостаточно.

Прежде чем загружать файл дампа, посмотрите на его размер и подумайте, сколько времени он будет обрабатываться и не надо ли что-то сделать перед началом процедуры, например уведомить пользователей или деактивировать часть приложения. Полезно на этот период отключить запись в двоичный журнал, если только вы не хотите реплицировать восстанавливаемые данные на подчиненный сервер: загружать огромный дамп серверу и так-то нелегко, а запись в двоичный журнал только увеличивает затраты (возможно, без всякой необходимости). Кроме того, в некоторых подсистемах хранения загрузка очень больших файлов требует особых предосторожностей. Например, не стоит загружать файл размером 100 Гбайт в таблицу InnoDB одной транзакцией, поскольку в результате образуется огромный сегмент отката. Необходимо разбить файл на порции и коммитить транзакцию после загрузки каждой из них.

Существует две разновидности восстановления в соответствии с двумя видами логических копий.

Загрузка SQL-файлов

SQL-дамп содержит исполняемые команды SQL. Вам остается только запустить его. Предположим, что демонстрационная база данных Sakila выгружена в один файл вместе со схемой. Тогда для восстановления данных обычно используется такая команда:

```
$ mysql < sakila-backup.sql
```

Можно также загрузить файл из клиента `mysql` командой `SOURCE`. Хотя это в общем-то лишь другой способ сделать то же самое, выполнить некоторые вещи при этом оказывается проще. Например, если вы обладаете административными правами в MySQL, то можете отключить запись в двоичный журнал команд, выполняемых в текущем соединении, а затем загрузить файл без перезапуска сервера:

```
mysql> SET SQL_LOG_BIN = 0;
mysql> SOURCE sakila-backup.sql;
mysql> SET SQL_LOG_BIN = 1;
```

Но при использовании команды `SOURCE` имейте в виду, что ошибка не прерывает выполнения пакета команд, как это происходит по умолчанию в случае чтения из стандартного ввода `mysql`.

Если резервная копия была сжата, то не нужно перед загрузкой отдельно распаковывать ее. Разархивирование и загрузку можно объединить в одной операции, что будет гораздо быстрее:

```
$ gunzip -c sakila-backup.sql.gz | mysql
```

О том, как загрузить сжатый файл командой `SOURCE`, будет рассказано при обсуждении именованных каналов в следующем разделе.

А вдруг нам понадобится вернуть данные только для одной таблицы (например, `actor`)? Если данные не содержат символов перехода на новую строку и схема таблицы уже присутствует, то восстановить такую таблицу нетрудно:

```
$ grep 'INSERT INTO `actor`' sakila-backup.sql | mysql sakila
```

Или для сжатого файла:

```
$ gunzip -c sakila-backup.sql.gz | grep 'INSERT INTO `actor`'| mysql sakila
```

Если нужно не только восстановить информацию, но и создать таблицу, а вся база данных находится в одном файле, то этот файл придется отредактировать. Именно поэтому многие предпочитают выгружать каждую таблицу в отдельный файл. Большинство редакторов не умеют работать с очень большими файлами, особенно сжатыми. Но вам, собственно, и не требуется редактировать файл, достаточно лишь извлечь из него нужные строки, поэтому подойдут утилиты командной строки. Довольно просто воспользоваться инструментом grep для выборки только команд `INSERT`, относящихся к данной таблице (как показано ранее), но извлечь команду `CREATE TABLE` сложнее. Ниже приведен скрипт sed, извлекающий ровно то, что необходимо:

```
$ sed -e '/./{H;$!d;}' -e 'x;/CREATE TABLE `actor`/!d;q' sakila-backup.sql
```

Да, признаем, выглядит загадочно. Если для восстановления данных вам приходится идти на подобные трюки, значит, процедура резервного копирования плохо продумана. Когда все распланировано заранее, можно будет избежать ситуаций, при которых вы в панике пытаетесь разобраться, как работает sed. Просто копируйте каждую таблицу в отдельный файл, а еще лучше — помещайте данные и схему в разные файлы.

Загрузка файлов с разделителями

При выгрузке данных командой `SELECT INTO OUTFILE` загружать их обратно следует командой `LOAD DATA INFILE` с теми же параметрами. Можно также воспользоваться утилитой mysqlimport, которая является не чем иным, как оберткой вокруг `LOAD DATA INFILE`. Куда загружать данные из файла, утилита mysqlimport определяет на основании соглашений об именовании.

Мы надеемся, что вы выгрузили не только данные, но и схему. В таком случае это SQL-дамп, и для его загрузки можно воспользоваться приемами из предыдущего пункта.

Для команды `LOAD DATA INFILE` существует замечательная оптимизация. Поскольку эта директива читает данные непосредственно из файла, то может возникнуть мысль предварительно его распаковать, а это очень длительная операция, которая к тому же активно обращается к диску. Но существует обходной путь, по крайней мере в системах, поддерживающих именованные каналы FIFO, к каковым относится и GNU/Linux. Сначала создайте именованный канал и направьте в него распакованные данные:

```
$ mkfifo /tmp/backup/default/sakila/payment.fifo
$ chmod 666 /tmp/backup/default/sakila/payment.fifo
$ gunzip -c /tmp/backup/default/sakila/payment.txt.gz
  > /tmp/backup/default/sakila/payment.fifo
```

Обратите внимание на употребление знака «больше» (`>`) для перенаправления распакованного потока в файл `payment.fifo` — нужен именно он, а не знак '`|`', который создает анонимный канал между программами. Мы используем `payment.fifo` — именованный канал, поэтому анонимный нам ни к чему.

Канал ждет, когда какая-нибудь программа откроет его и начнет читать данные с другого конца. В этом-то и заключается изюминка: сервер MySQL может получать распакованные данные из канала точно так же, как из обычного файла. Не забудьте отключить запись в двоичный журнал, если она не нужна:

```
mysql> SET SQL_LOG_BIN = 0; -- Необязательно
    -> LOAD DATA INFILE '/tmp/backup/default/sakila/payment.fifo'
    -> INTO TABLE sakila.payment;
Query OK, 16049 rows affected (2.29 sec)
Records: 16049  Deleted: 0  Skipped: 0  Warnings: 0
```

После того как MySQL закончит загрузку данных, утилита gunzip завершится и именованный канал можно будет удалить. Такую же технику можно применять для загрузки сжатых файлов из командного клиента mysql директивой `SOURCE`. Программа pt-fifo-split

в пакете Percona Toolkit может помочь загружать большие файлы не за одну большую транзакцию, а партиями, что может оказаться намного более эффективным.

Вы не можете добраться туда отсюда

Однажды один из авторов этой книги поменял тип столбца с `DATETIME` на `TIMESTAMP`, чтобы сэкономить место и ускорить обработку, как рекомендовано в главе 3. В результате определение таблицы стало выглядеть так:

```
CREATE TABLE tbl (
    col1 timestamp NOT NULL,
    col2 timestamp NOT NULL default CURRENT_TIMESTAMP
         on update CURRENT_TIMESTAMP,
    ... прочие столбцы ...
);
```

Такое определение таблицы вызывает синтаксическую ошибку в версии MySQL 5.0.40, в которой оно было создано. Выгрузить данные удастся, а загрузить обратно — нет. Вот из-за таких странных непредвиденных ошибок и надо проверять резервные копии. Никогда заранее не знаешь, что помешает вернуть данные!

Восстановление на конкретный момент времени

Самый распространенный способ восстановления на конкретный момент времени в MySQL заключается в том, чтобы вернуть данные с последней полной резервной копии, а затем воспроизвести двоичные журналы, накопившиеся за этот период (иногда это называется накатом журналов). Имея двоичный журнал, можно восстановиться на любой момент времени. Можно даже без особого труда восстановить только одну базу данных.

Главный недостаток процесса заключается в том, что воспроизведение двоичного журнала может выполняться медленно. Оно, по сути, эквивалентно репликации. Если у вас есть подчиненный сервер и вы измерили степень использования потока SQL, то узнаете, как скоро можно будет воспроизвести двоичные журналы. Например, если поток SQL используется примерно на 50 %, восстановление двоичных журналов за неделю может занять от трех до четырех дней.

Нередко возникает задача отменить действие необдуманно выполненной команды, например `DROP TABLE`. Рассмотрим упрощенный пример, на котором покажем, как это сделать, когда имеются только таблицы типа MyISAM.

Предположим, что в полночь задание резервного копирования выполнило следующие команды, которые копируют базу данных в другое место на том же сервере:

```
mysql> FLUSH TABLES WITH READ LOCK;
    -> server1# cp -a /var/lib/mysql/sakila /backup/sakila;
mysql> FLUSH LOGS;
    -> server1# mysql -e "SHOW MASTER STATUS" --vertical > /backup/master.info;
mysql> UNLOCK TABLES;
```

Допустим, что позже в тот же день кто-то выполнил следующую команду:

```
mysql> USE sakila;
mysql> DROP TABLE sakila.payment;
```

Для простоты предположим, что мы можем корректно восстановить эту базу данных отдельно от других (то есть в этой базе не существует таблиц, которые были вовлечены в запросы, затрагивающие несколько баз данных). Предположим также, что ошибочная команда была обнаружена спустя некоторое время. Наша задача — восстановить все, что происходило с этой базой данных, за исключением вышеупомянутой команды.

Иными словами, мы должны сохранить все обновления в других таблицах, в том числе выполненные после злополучной команды.

Это не так уж трудно сделать. Прежде всего остановим MySQL, чтобы предотвратить дальнейшие модификации, и вернем из резервной копии только базу данных Sakila:

```
server1# /etc/init.d/mysql stop
server1# mv /var/lib/mysql/sakila /var/lib/mysql/sakila.tmp
server1# cp -a /backup/sakila /var/lib/mysql
```

Запретим обычные соединения, временно добавив в файл `my.cnf` такие строчки:

```
skip-networking
socket=/tmp/mysql_recover.sock
```

Теперь можно без опаски запустить сервер:

```
server1# /etc/init.d/mysql start
```

Следующая задача — найти в двоичном журнале те команды, которые мы хотим воспроизвести, и те, которые следует пропустить. Как выясняется, после полуночи сервер создал только один двоичный журнал. Мы можем обработать его утилитой grep и найти ошибочно выполненную команду:

```
server1# mysqlbinlog --database=sakila /var/log/mysql/mysql-bin.000215
| grep -B3 -i 'drop table sakila.payment'
 # at 352
#070919 16:11:23 server id 1  end_log_pos 429  Query  thread_id=16  exec_time=0
error_code=0
SET TIMESTAMP=1190232683/*!*/;
DROP TABLE sakila.payment/*!*/;
```

Итак, команда, которую мы хотим пропустить, начинается с позиции 352 в файле журнала, а следующая за ней — с позиции 429. Воспроизведем журнал до позиции 352 и далее с позиции 429 до конца:

```
server1# mysqlbinlog --database=sakila /var/log/mysql/mysql-bin.000215
 --stop-position=352 | mysql -uroot -p
server1# mysqlbinlog --database=sakila /var/log/mysql/mysql-bin.000215
 --start-position=429 | mysql -uroot -p
```

Теперь осталось только еще раз проверить данные, остановить сервер, вернуть файл `my.cnf` в исходное состояние и снова запустить сервер.

Более сложные методы восстановления

В основе репликации и восстановления на конкретный момент времени лежит один и тот же механизм — двоичный журнал. Это означает, что репликация может оказаться полезным, хотя и не вполне очевидным, инструментом восстановления. В этом разделе мы продемонстрируем некоторые возможности. Список неполон, но содержит некоторые мысли о том, как спроектировать процедуру восстановления с учетом своих потребностей. Еще раз напомним — оформляйте все, что может понадобиться в процессе восстановления, в виде скрипта и прогоняйте его на тестовом сервере.

Быстрое восстановление с помощью отложенной репликации

Ранее в этой главе мы уже отмечали, что наличие подчиненного сервера, реплицирующего данные с задержкой, может существенно упростить и ускорить восстановление на конкретный момент времени, если вы успели заметить случайную ошибку до того, как подчиненный сервер выполнил породившую ее команду.

Эта процедура немного отличается от описанной в предыдущем разделе, но идея та же самая. Останавливаем подчиненный сервер, а затем применяем команду `START SLAVE UNTIL`, чтобы воспроизвести события, предшествующие команде, которую нужно пропустить. Затем выполняем команду `SET GLOBAL SQL_SLAVE_SKIP_COUNTER=1`, чтобы пропустить ошибочную директиву. Если нужно проигнорировать несколько событий, задайте значение больше 1 или просто воспользуйтесь командой `CHANGE MASTER`, чтобы продвинуть вперед позицию, с которой подчиненный сервер читает журнал.

Осталось только выполнить команду `START SLAVE` и дать подчиненному серверу возможность закончить обработку своих журналов повтора. В результате подчиненный сервер проделает за вас всю утомительную работу по восстановлению на конкретный момент времени. Затем делаем подчиненный сервер главным — и восстановление закончено, причем почти без прерывания обслуживания.

Даже если подчиненного сервера с отложенной репликацией нет, все равно такие серверы могут быть полезны, потому что копируют двоичные журналы главного сервера на другую машину. Если диск на главном сервере выйдет из строя, то журналы повтора на подчиненном могут оказаться единственным местом, где имеются сравнительно актуальные двоичные журналы главного сервера.

Восстановление с помощью сервера журналов

Существует еще один способ применить репликацию для восстановления: настроить сервер журналов. Мы считаем, что репликация более надежна, чем утилита mysqlbinlog, в которой могут встретиться странные ошибки и которая к тому же иногда ведет себя неожиданно. Кроме того, использование сервера журналов — более гибкий и простой путь восстановления, чем работа с утилитой mysqlbinlog, не только

из-за наличия команды START SLAVE UNTIL, но и из-за действующих правил репликации (например, replicate-do-table). При наличии сервера журналов можно реализовать гораздо более сложные правила фильтрации, чем без него.

Например, сервер журналов позволяет без труда восстановить одну таблицу. С помощью mysqlbinlog и утилит командной строки сделать это значительно труднее — настолько, что мы не советуем даже пытаться.

Предположим, что наш беспечный разработчик удалил ту же таблицу, что и раньше, а нам нужно восстановить ее, не возвращая весь сервер в то состояние, в котором он находился в момент снятия ночной копии. Вот как можно это сделать с помощью сервера журналов.

1. Пусть сервер, подлежащий восстановлению, называется server1.
2. Восстановим ночную копию на другом сервере — server2. Запустим на нем процедуру восстановления, чтобы не ухудшить ситуацию в случае, если в ходе этого процесса будет допущена ошибка.
3. Настроим сервер журналов так, чтобы он обслуживал двоичные журналы сервера server1, следуя указаниям, приведенным в главе 10 (было бы неплохо для страховки скопировать журналы на какой-нибудь другой сервер и настроить сервер журналов там).
4. Изменим конфигурационный файл server2, включив в него следующую строку:

   ```
   replicate-do-table=sakila.payment
   ```

5. Перезапустим server2 и сделаем его подчиненным серверу журналов с помощью команды CHANGE MASTER TO. Сконфигурируем его так, чтобы он читал двоичный журнал с позиции, соответствующей ночной копии. Но пока не будем выполнять команду START SLAVE.
6. Выполним на сервере server2 команду SHOW SLAVE STATUS и убедимся, что все правильно. Семь раз отмерь, один раз отрежь!
7. Найдем в двоичном журнале место, с которого начинается ошибочная команда, и выполним команду START SLAVE UNTIL, чтобы воспроизвести все события вплоть до этой позиции.
8. Остановим процесс репликации на сервере server2 командой STOP SLAVE. Теперь таблица существует в виде, непосредственно предшествующем виду в момент удаления.
9. Скопируем таблицу с server2 на server1.

Все это возможно только в том случае, если данная таблица не участвует в командах UPDATE, DELETE или INSERT, которые выполняют многотабличные обновления. Любая такая команда будет выполняться в состоянии базы данных, отличном от того, которое существовало в момент записи событий в двоичный журнал, поэтому, скорее всего, после восстановления таблица будет содержать не ту информацию, которую должна бы. (Это справедливо только в том случае, если вы используете покомандное двоичное журналирование, если же задействовано построчное журналирование, процесс восстановления не будет подвержен возникновению этой ошибки.)

Восстановление InnoDB

InnoDB проверяет файлы данных и журналов при каждом запуске, чтобы понять, нужно ли начинать процедуру восстановления. Однако восстановление InnoDB — не то же самое, о чем мы говорим в этой главе. Речь идет не о воссоздании данных из резервной копии, а о применении к файлам данных тех транзакций, которые хранятся в журнале, и об откате незакоммиченных транзакций.

Детали процедуры восстановления InnoDB слишком сложны, чтобы описывать их здесь. Вместо этого расскажем о том, как выполнить эту процедуру на практике, когда в работе InnoDB возникает серьезная проблема.

Обычно InnoDB прекрасно исправляет ошибки самостоятельно. Если только в самом коде MySQL нет дефекта и оборудование исправно, то ничего экстраординарного вам делать не придется даже после отключения питания сервера. InnoDB просто произведет штатное восстановление при запуске, и все будет хорошо. В журнале ошибок вы увидите примерно такие сообщения:

```
InnoDB: Doing recovery: scanned up to log sequence number 0 40817239
InnoDB: Starting an apply batch of log records to the database...
```

InnoDB выводит в журнал сообщения о ходе восстановления (сколько уже выполнено в процентах). Некоторые пользователи жаловались, что не видят никаких сообщений, пока процесс не завершится. Но будьте терпеливы, ускорить эту процедуру невозможно. Если вы принудительно завершите ее и начнете заново, то лишь продлите ожидание.

Если возникла серьезная неисправность оборудования, скажем, поврежден диск или память, или же вы наткнулись на ошибку в коде MySQL или InnoDB, то, возможно, придется вмешаться и либо принудительно запустить восстановление, либо отменить штатную процедуру.

Причины повреждения InnoDB

Вообще говоря, подсистема InnoDB очень надежна. Она проектировалась именно с такой, в нее встроено множество проверок, направленных на то, чтобы предотвратить порчу данных, а также найти и исправить их повреждение, — гораздо больше, чем в некоторых других подсистемах хранения. Однако она не может защитить себя от всего на свете.

Как минимум InnoDB полагается на то, что обращения к небуферизованному вводу/выводу и системному вызову `fsync()` не возвращают управление, пока данные не будут окончательно записаны на физический носитель. Если оборудование не гарантирует этого, то InnoDB не сможет защитить информацию и сбой приведет к ее повреждению.

Многие проблемы, связанные с порчей данных в таблицах InnoDB, обусловлены работой оборудования, например неправильной записью на страницу из-за сбоя питания или ошибки памяти. Но наш опыт показывает, что гораздо чаще причиной

является некорректная конфигурация устройств. К числу типичных ошибок можно отнести включение кэша записи на карте RAID-контроллера без резервного питания или включение кэша записи в самих накопителях на жестких дисках. Из-за таких ошибок контроллер лжет, сообщая, что вызов `fsync()` завершен, тогда как на самом деле данные еще находятся в кэше записи, а не на диске. Другими словами, оборудование не дает гарантий, необходимых InnoDB для безопасного хранения данных.

Иногда компьютеры конфигурируются так по умолчанию, поскольку этот режим обеспечивает более высокую производительность. Для каких-то целей это, может быть, хорошо, но только не для сервера транзакционной базы данных.

Повреждения возможны и тогда, когда InnoDB работает с хранилищем, подключенным по сети (NAS), поскольку завершение системного вызова `fsync()` в этом случае означает лишь, что устройство получило данные. Сами данные будут в безопасности, если выйдет из строя InnoDB, но могут оказаться испорчены в случае отказа NAS-устройства.

Степень повреждения может быть различной. Серьезные ошибки способны вызвать аварийный останов InnoDB или сервера MySQL, а менее серьезные означают просто потерю некоторых транзакций из-за того, что файл журнала не был сброшен на диск.

Как восстановить поврежденные данные InnoDB

Существует три основных типа повреждений InnoDB, и в каждом случае для восстановления применяются различные методы.

- *Повреждение вторичного индекса.* Исправить повреждение вторичного индекса часто удается с помощью команды `OPTIMIZE TABLE`. Как альтернативу можно использовать команду `SELECT INTO OUTFILE`, удалить и заново создать таблицу, а затем загрузить в нее данные командой `LOAD DATA INFILE`. (Также можно перевести таблицу в подсистему хранения MyISAM и обратно.) Повреждение устраняется за счет того, что строится новая таблица и, следовательно, испорченный индекс тоже перестраивается.
- *Повреждение кластерного индекса.* Возможно, нужно будет воспользоваться параметром `innodb_force_recovery`, чтобы выгрузить дамп таблицы (подробнее об этом далее). Иногда в процессе выгрузки InnoDB аварийно завершает работу, в этом случае приходится выгружать таблицу по частям, чтобы обойти поврежденные страницы, вызывающие аварию сервера. Повреждение кластерного индекса серьезнее, чем вторичного, поскольку затронутыми оказываются сами строки с данными, но во многих случаях поврежденные таблицы все-таки можно исправить.
- *Повреждение системных структур.* К системным структурам относятся журнал транзакций InnoDB, область отмены в табличном пространстве и словарь данных.

Для исправления таких повреждений, скорее всего, потребуется произвести полную выгрузку и восстановление, потому что затронутыми могут оказаться многие внутренние механизмы работы InnoDB.

Обычно поврежденный вторичный индекс удается исправить без потери данных. Но в остальных двух случаях утрата какой-то части информации вполне вероятна. При наличии резервной копии лучше вернуть данные с нее, а не пытаться извлечь их из поврежденных файлов.

Если никакой альтернативы извлечению данных из поврежденных файлов нет, то сначала нужно запустить InnoDB, а потом выполнить команду `SELECT INTO OUTFILE` для выгрузки информации в файл. Если сервер аварийно завершил работу и не удается даже запустить InnoDB, то можно сконфигурировать его так, чтобы не загружались штатные процессы восстановления и фоновые потоки. Возможно, после этого удастся стартовать и создать логическую копию с ослабленным контролем целостности или вовсе без него.

Параметр `innodb_force_recovery` управляет тем, какие операции InnoDB выполняет при запуске и во время нормальной работы. Обычно он равен `0`, но его можно увеличить до `6`. В руководстве по MySQL описано поведение сервера в каждом из режимов. Мы не будем здесь дублировать эту информацию, но отметим, что значения вплоть до `4` не представляют особой опасности. При таких настройках можно лишь потерять некоторые данные на поврежденных страницах, но если пойти дальше, то не исключено извлечение некорректной информации из поврежденных страниц. Также повышается риск аварийного завершения во время выполнения команды `SELECT INTO OUTFILE`. Иными словами, уровни с нулевого до четвертого не наносят вреда данным, но InnoDB может не воспользоваться некоторыми возможностями их исправления, на уровнях 5 и 6 попытки ликвидировать повреждения становятся более настойчивыми, но это сопряжено с риском нанести ущерб данным.

Если параметр `innodb_force_recovery` больше `0`, то InnoDB, по существу, работает в режиме чтения, но создавать и удалять таблицы все же можно. Это предотвращает развитие повреждений и заставляет InnoDB немного смягчить обычные проверки и не прекращать работу немедленно в случае обнаружения испорченных данных. В обычных условиях принудительное завершение работы при обнаружении испорченной информации служит предохранителем, но во время восстановления это нежелательно. Если нужно заставить InnoDB выполнить восстановление, то имеет смысл сконфигурировать MySQL так, чтобы обычные подключения отвергались до завершения процедуры.

Если данные InnoDB повреждены настолько сильно, что MySQL вообще не запускается, то можно воспользоваться комплектом инструментов Percona InnoDB Recovery Toolkit, чтобы извлечь информацию напрямую из страниц табличного пространства. Эти инструменты можно бесплатно скачать с сайта http://www.percona.com/software/. Percona Server может работать даже тогда, когда некоторые таблицы повреждены, что отличает его от поведения MySQL по умолчанию, при котором при обнаружении одной поврежденной страницы происходит останов всего сервера.

Инструменты резервного копирования и восстановления

В настоящий момент существует множество хороших и не очень инструментов резервного копирования. Мы предпочитаем mylvm-backup для изготовления резервных копий снимков LVM и Percona XtraBackup (с открытым исходным кодом) или MySQL Enterprise Backup (запатентовано) для горячего резервного копирования InnoDB. Рекомендуем не пользоваться утилитой mysql-dump для резервного копирования значительного объема данных из-за перегрузки сервера и непредсказуемо длительного времени восстановления.

Существует несколько инструментов резервного копирования, которые давно должны были бы исчезнуть, но, к сожалению, не исчезли. Наиболее очевидным примером является mk-parallel-dump из пакета Maatkit, который никогда не работал правильно, хотя его несколько раз перепроектировали. Другим примером является mysqlhotcopy, который вроде бы раньше работал в таблицах MyISAM. Ни одному из этих инструментов полностью доверять нельзя, хотя они могут заставить вас поверить в то, что данные резервируются, когда этого не происходит. Например, mysqlhotcopy будет копировать `.ibd`-файлы, если вы используете InnoDB с параметром `innodb_file_per_table`, что заставит некоторых поверить (и напрасно), что InnoDB выполнила резервное копирование данных. В некоторых случаях оба инструмента могут неблагоприятно повлиять на работающий сервер.

Если вы видели планы MySQL в 2008 или 2009 году, то, вероятно, слышали об онлайновом резервном копировании MySQL. Речь шла о возможности, позволяющей запускать резервное копирование и восстановление с сервера с помощью команд SQL. Первоначально она была запланирована для MySQL 5.2, позже перенесена на MySQL 6.0, а затем, насколько нам известно, отменена совсем.

MySQL Enterprise Backup

Этот инструмент, ранее известный как InnoDB Hot Backup, или ibbackup, является частью подписки на MySQL Enterprise от Oracle. Его использование не требует остановки MySQL, установки блокировок или прерывания нормальной работы базы данных, хотя и вызовет некоторое увеличение нагрузки на ваш сервер. Он поддерживает такие возможности, как изготовление сжатых резервных копий, инкрементное резервное копирование и потоковое резервное копирование на другой сервер. Это официальный инструмент резервного копирования для MySQL.

Percona XtraBackup

Инструмент Percona XtraBackup во многом похож на MySQL Enterprise Backup, но у него открытый исходный код и он бесплатный. В дополнение к основному инструменту резервного копирования здесь есть также скрипт-обертка, написанный на Perl, который улучшает функциональность и позволяет решать более сложные

задачи. Он поддерживает потоковые, инкрементные, сжатые и многопоточные (параллельные) операции резервного копирования. А также имеет множество специальных возможностей для уменьшения влияния резервного копирования на сильно загруженные системы.

Percona XtraBackup работает с хвостом файлов журнала InnoDB в фоновом потоке, а затем копирует файлы данных InnoDB. Это слегка запутанный процесс со специальными проверками для обеспечения постоянного копирования данных. Когда все файлы данных копируются, останавливается поток, копирующий журналы. Результатом является копия всех данных, но в разные моменты времени. Теперь журналы могут быть применены к файлам данных с использованием процедур аварийного восстановления InnoDB, что должно привести все файлы данных в согласованное состояние. Это называется процессом подготовки. После подготовки резервная копия полностью согласована и содержит все закоммиченные транзакции на момент завершения процесса копирования файлов. Все эти процессы происходят вне MySQL, поэтому серверу не нужно каким-либо образом подключаться к MySQL или обращаться к ней.

Скрипт-обертка добавляет возможность восстановления резервной копии, скопировав ее обратно в исходное местоположение. Кроме того, существует проект Lachlan Mulcahy XtraBackup Manager с еще большей функциональностью (для получения дополнительной информации см. http://code.google.com/p/xtrabackup-manager/).

mylvmbackup

Написанный Ленцем Гриммером (Lenz Grimmer) на языке Perl скрипт mylvmbackup (http://lenz.homelinux.org/mylvmbackup/) призван автоматизировать создание резервных копий MySQL с помощью снимков LVM. Он получает глобальную блокировку чтения, создает снимок, а затем снимает блокировку. После этого данные упаковываются с помощью программы tar и снимок удаляется. Архиву присваивается имя, основанное на времени создания копии. У инструмента есть несколько дополнительных параметров, но в целом он предназначен просто для выполнения резервного копирования LVM.

Zmanda Recovery Manager

Программа Zmanda Recovery Manager для MySQL (ZRM) (http://www.zmanda.com) распространяется в виде бесплатной (на условиях лицензии GPL) и коммерческой версий. В состав редакции для предприятия (Enterprise Edition) входят консоль управления, реализующая графический веб-интерфейс конфигурирования, резервного копирования, верификации, восстановления, генерации отчетов и запуска по расписанию. Версия с открытым исходным кодом никоим образом не урезана, однако она не включает в себя некоторые дополнительные тонкости, такие как веб-консоль.

Как и следует из ее названия, ZRM на самом деле является менеджером резервного копирования и восстановления, а не просто инструментом. Он обертывает

собственную функциональность вокруг стандартных инструментов и методов, таких как mysqldump, снимки LVM и Percona XtraBackup. ZRM автоматизирует бóльшую часть серьезной работы по резервному копированию и восстановлению.

mydumper

Несколько нынешних и бывших инженеров, работавших над MySQL, используя накопленный за многие годы опыт, создали утилиту mydumper в качестве замены mysqldump. Это многопоточный (параллельный) набор инструментов для резервного копирования и восстановления для MySQL и Drizzle с множеством приятных функций. Многие, по-видимому, посчитают скорость многопоточного резервного копирования и восстановления основным достоинством этого инструмента. Мы знаем тех, кто использует его в работе, однако у нас самих такого опыта нет. Больше подробностей об этом инструменте вы можете получить на http://www.mydumper.org.

mysqldump

Большинство людей используют программы, которые поставляются с MySQL, поэтому, несмотря на свои недостатки, наиболее распространенным вариантом для создания логических резервных копий данных и схем является mysqldump. Это инструмент общего назначения, пригодный для решения многих задач, таких как копирование таблицы с одного сервера на другой.

```
$ mysqldump --host=server1 test t1 | mysql --host=server2 test
```

В этой главе мы уже приводили примеры создания логических копий с помощью mysqldump. По умолчанию эта программа выводит скрипт, содержащий все команды, необходимые для создания таблицы и заполнения ее данными, существуют также флаги для вывода представлений, хранимых процедур и триггеров. Вот несколько типичных примеров ее использования.

- Создание логической копии всего, что хранится на сервере, в виде одного файла, со всеми таблицами во всех базах данных в один и тот же момент времени:

  ```
  $ mysqldump --all-databases > dump.sql
  ```

- Создание логической копии базы данных Sakila:

  ```
  $ mysqldum8p --databases sakila > dump.sql
  ```

- Создание логической копии только таблицы `sakila.actor`:

  ```
  $ mysqldump sakila actor > dump.sql
  ```

Флаг `--result-file` позволяет задать выходной файл, что предотвращает преобразование символов перехода на новую строку на платформе Windows:

```
$ mysqldump sakila actor --result-file=dump.sql
```

Параметры mysqldump, подразумеваемые по умолчанию, не годятся для серьезного резервного копирования. Было бы правильно явно задать некоторые флаги. Далее

перечислены флаги, которыми мы часто пользуемся для того, чтобы повысить эффективность mysqldump и упростить работу с генерируемыми файлами.

- `--opt`. Относится к группе параметров, отключающих буферизацию, которая может привести к исчерпанию памяти сервера, порождает меньше SQL-команд в дампе, так что они загружаются более эффективно, и делает ряд других полезных вещей. Подробную информацию можно почерпнуть из справки. Если эту группу параметров отключить, то mysqldump будет сохранять каждую выгружаемую таблицу в памяти перед тем, как записать ее на диск, для больших таблиц это непрактично.
- `--allow-keywords`, `--quote-names`. Позволяет выгружать и загружать таблицы, имена которых совпадают с зарезервированными словами.
- `--complete-insert`. Позволяет перемещать данные между таблицами с неодинаковыми столбцами.
- `--tz-utc`. Позволяет перемещать данные между серверами, находящимися в разных часовых поясах.
- `--lock-all-tables`. Выполняет команду `FLUSH TABLES WITH READ LOCK` для получения глобально согласованной резервной копии.
- `--tab`. Формирует файлы дампов с помощью команды `SELECT INTO OUTFILE`.
- `--skip-extended-insert`. Выводит каждую строку данных в виде отдельной команды `INSERT`. Это позволит при необходимости избирательно восстанавливать некоторые строки. Но в таком случае большие файлы будут импортироваться в MySQL дольше, поэтому задавать флаг нужно с осторожностью.

При задании флагов `--databases` или `--all-databases` сформированные для каждой базы дампы будут содержать согласованные данные, потому что mysqldump блокирует и выгружает все таблицы в базе одновременно. Однако таблицы из разных баз все же могут оказаться рассогласованными. Флаг `--lock-all-tables` решает и эту проблему.

Для резервного копирования InnoDB следует добавить параметр `--single-transaction`, который использует возможности MVCC InnoDB для создания согласованной резервной копии в единый момент времени вместо применения `LOCK TABLES`. Если вы добавите параметр `--master-data`, резервная копия будет содержать также координаты двоичного журнала сервера в момент резервного копирования, что очень полезно для восстановления на конкретный момент времени и настройки подчиненных серверов. Однако имейте в виду, что она будет использовать команду `FLUSH TABLES WITH READ LOCK`, чтобы приостановить работу сервера и дать ему возможность получить координаты.

Скрипты резервного копирования

Часто возникает необходимость в написании скрипта для резервного копирования. Вместо того чтобы приводить пример демонстрационной программы, что только заняло бы место на страницах книги, мы перечислим составные части типичного скрипта и покажем несколько фрагментов на языке Perl. Можете считать их кирпичиками,

из которых можно собрать свой собственный сценарий. Приведем их примерно в том порядке, в котором они используются.

- *Проверка корректности.* Облегчите жизнь себе и своим коллегам — включите строгий контроль ошибок и используйте английские имена переменных:

```
use strict:
use warnings FATAL => 'all';
use English qw(-no_match_vars);
```

Если скрипт пишется на языке оболочки bash, то можно также включить строгий контроль переменных. Следующие команды заставляют интерпретатор выдавать соответствующее сообщение, если в подстановке встретится неопределенная переменная или какая-то программа завершится с ошибкой:

```
set -u;
set -e;
```

- *Аргументы командной строки.* Лучший способ добавить обработку параметров командной строки — это воспользоваться стандартными библиотеками, которые входят в любую установку языка Perl:

```
use Getopt::Long;
Getopt::Long::Configure('no_ignore_case', 'bundling');
GetOptions( .... );
```

- *Подключение к серверу MySQL.* Стандартный Perl-модуль DBI доступен практически везде и обеспечивает необходимые выразительность и гибкость. О том, как с ним работать, говорится в документации, сгенерированной программой perldoc (имеется на сайте http://search.cpan.org). Подключиться к MySQL, используя DBI, можно следующим образом:

```
use DBI;
$dbh = DBI->connect(
    'DBI:mysql:;host=localhost', 'user', 'pass', {RaiseError => 1 });
```

О написании скриптов командной строки прочитайте в оперативной справке, которую программа mysql выводит при задании флага `--help`. У нее масса параметров, облегчающих использование в составе скриптов. Вот, например, как можно перебрать список баз данных в bash-скрипте:

```
mysql -ss -e 'SHOW DATABASES' | while read DB; do
    echo "${DB}"
done
```

- *Останов и запуск MySQL.* Самый лучший способ остановить и запустить сервер MySQL — воспользоваться методом, принятым в вашей операционной системе, например выполнить скрипт инициализации `/etc/init.d/mysql` или задействовать диспетчер служб (для Windows). Но это не единственный способ. Можно остановить СУБД из Perl-скрипта, имея какое-нибудь соединение с базой данных:

```
$dbh->func("shutdown", 'admin');
```

Не следует считать, что MySQL будет остановлена после того, как эта команда завершится, — возможно, процесс останова все еще идет. Можно также остановить MySQL из командной строки:

```
$ mysqladmin shutdown
```

- *Получение списков баз данных и таблиц.* Любой скрипт резервного копирования запрашивает у MySQL список баз данных и таблиц. Помните, что в этот список могут входить и ненастоящие базы данных, например каталог `lost+found` в некоторых журналирующих файловых системах, а также `INFORMATION_SCHEMA`. Убедитесь, что ваш скрипт правильно обрабатывает представления, и не забывайте, что выполнение команды `SHOW TABLE STATUS` может занять много времени, когда вы имеете дело с большими объемами данных в таблицах типа InnoDB:

```
mysql> SHOW DATABASES;
mysql> SHOW /*!50002 FULL*/ TABLES FROM <database>;
mysql> SHOW TABLE STATUS FROM <database>;
```

- *Блокировка, сброс и разблокировка таблиц.* Иногда необходимо выполнять блокировку и/или сброс одной или нескольких таблиц. Можно либо заблокировать все нужные таблицы, перечислив их имена, либо захватить глобальную блокировку:

```
mysql> LOCK TABLES <database.table> READ [, ...];
mysql> FLUSH TABLES;
mysql> FLUSH TABLES <database.table> [, ...];
mysql> FLUSH TABLES WITH READ LOCK;
mysql> UNLOCK TABLES;
```

  Не забывайте о возможной гонке при получении списков таблиц и их блокировке. Могут быть созданы новые таблицы, а старые удалены или переименованы. Если вы блокируете и копируете таблицы поодиночке, то согласованной копии не получится.

- *Сброс двоичных журналов.* Очень полезно попросить сервер начать новый двоичный журнал (делайте это после блокировки таблиц, но перед тем, как начинать резервное копирование):

```
mysql> FLUSH LOGS;
```

  Восстановление и инкрементное резервное копирование будет проще выполнить, если не придется думать о том, с какого места в середине файла журнала начинать. Но возможны побочные эффекты, связанные со сбросом и открытием новых журналов, а кроме того, есть вероятность уничтожить старые записи, поэтому внимательно следите за тем, чтобы не отбросить данные, которые еще могут понадобиться.

- *Получение позиции в двоичном журнале.* Скрипт должен получить и куда-то записать состояние как главного, так и подчиненного сервера, даже если данный сервер выступает только в роли главного или только в роли подчиненного:

```
mysql> SHOW MASTER STATUS\G
mysql> SHOW SLAVE STATUS\G
```

  Выполните обе команды и игнорируйте возможные ошибки, чтобы скрипт получил всю возможную информацию.

- *Выгрузка данных.* Лучше всего использовать утилиту mysqldump и команду `SELECT INTO OUTFILE`.

- *Копирование данных.* Используйте какой-либо из методов, рассмотренных в этой главе.

Это кирпичики, из которых состоит любой скрипт резервного копирования. Самое трудное — написать скрипт восстановления. Если вам хочется узнать, как сделать это правильно, изучите исходный код системы ZRM.

Итоги главы

Все знают, что резервные копии нужны, но не все понимают, что им нужны восстанавливаемые резервные копии. Существует множество способов создания резервных копий, которые противоречат требованиям к восстановлению. Чтобы избежать этой проблемы, стоит определить и задокументировать целевую точку восстановления и целевое время восстановления и использовать их как ориентиры при выборе системы резервного копирования.

Также важно регулярно тестировать процесс восстановления, чтобы быть уверенными в том, что он работает. Очень просто настроить утилиту mysqldump и позволить ей работать каждую ночь, не понимая, что объем данных со временем разрастется до такой степени, что для их импорта может потребоваться несколько дней или даже недель. Хуже всего обнаружить, сколько времени требуется на восстановление, тогда, когда вам оно действительно необходимо. Резервная копия, на создание которой затрачиваются часы работы, может потребовать в прямом смысле нескольких недель для восстановления в зависимости от используемого оборудования, схемы, индексов и данных.

Не попадите в ловушку, посчитав, что подчиненный сервер — это резервная копия. Это менее разрушающий источник для резервного копирования, но не резервная копия. То же самое можно сказать о RAID-томе, SAN и снимках файловой системы. Убедитесь, что ваши резервные копии могут пройти тест DROP TABLE (или тест «Меня взломали»), а также тест потери центра обработки данных. Если вы берете резервные копии с подчиненного сервера, убедитесь, что проверяете целостность репликации с помощью утилиты pt-table-checksum.

Два наших любимых способа резервного копирования — копирование данных из файловой системы или снимков SAN или использование Percona XtraBackup. Оба они позволяют создавать неразрушающие двоичные (физические) резервные копии ваших данных, которые затем можно верифицировать, запустив экземпляр mysqld и проверяя таблицы.

Иногда вы даже можете одним выстрелом убить двух зайцев: каждый день проверять восстановление данных, возвращая их из резервной копии на разработку или обкатку на сервер. Вы также можете сбросить данные из этого экземпляра, чтобы создать логическую резервную копию. Мы считаем необходимым наряду с этим создавать резервные копии двоичных журналов и хранить достаточное количество резервных копий и двоичных журналов разных версий, благодаря которым можно выполнить восстановление или настроить новый подчиненный сервер, даже если последняя резервная копия непригодна для использования.

Помимо продуктов с открытым исходным кодом, о которых мы упоминали, существуют хорошие коммерческие инструменты резервного копирования, прежде всего MySQL Enterprise Backup. Будьте осторожны с инструментами резервного копирования, которые входят в состав графического редактора SQL-запросов, инструментами управления сервером и т. п. А также с плагинами резервного копирования MySQL от компаний, которые создают один на все случаи жизни инструмент резервного копирования, поддерживающий в том числе и MySQL. На самом деле вам нужен отличный инструмент резервного копирования, предназначенный в первую очередь для MySQL, а не тот, который просто поддерживает MySQL и сотни других продуктов. Многие поставщики инструментов резервного копирования не знают возможностей таких методов, как использование `FLUSH TABLES WITH READ LOCK`, или не могут их верно оценить. На наш взгляд, использование этой команды SQL однозначно не позволяет считать такое резервное копирование горячим. Если вы используете только таблицы InnoDB, вам это обычно не нужно.

16 Инструменты для пользователей MySQL

В дистрибутив сервера MySQL не входят инструменты для решения многих типичных задач, например мониторинга сервера или сравнения данных, хранящихся на разных серверах. К счастью, Oracle предлагает ряд полезных коммерческих инструментов, расширяющих эти возможности. Кроме того, сообщество, сложившееся вокруг MySQL, разработало множество полезных утилит, избавляя вас от необходимости придумывать собственные.

Средства организации интерфейса

Инструменты из этой категории помогают запускать запросы, создавать таблицы и новых пользователей, а также решать другие рутинные задачи. В этом разделе мы приведем краткий перечень наиболее популярных приложений такого рода. Как правило, все или большинство их функций можно реализовать с помощью запросов или SQL-команд, они лишь помогают работать быстрее и с большим комфортом, допуская меньше ошибок

- *MySQL Workbench*. Это инструмент «всё в одном» для таких задач, как управление сервером, запись запросов, разработка хранимых процедур и работа со схемами. Он оснащен подключаемый интерфейсом, который позволяет вам писать собственные инструменты и интегрировать их в рабочую среду. Кроме того, существуют скрипты и библиотеки языка Python, использующие этот подключаемый интерфейс. Существует как бесплатная, так и коммерческая версия MySQL Workbench, при этом в последней имеются некоторые дополнительные возможности. Однако бесплатной версии более чем достаточно для большинства потребностей. Вы можете узнать больше на http://www.mysql.com/products/workbench/.
- *SQLyog*. Один из самых популярных визуальных инструментов для MySQL с несколькими отличными возможностями. Это инструмент того же класса, что и MySQL Workbench, однако каждый из них обладает некоторыми возможностями, отсутствующими у другого. Реализован он только на платформе Microsoft Windows, причем полная версия платная, а издание с ограниченной функциональностью распространяется бесплатно. Дополнительную информацию о программе SQLyog см. на сайте http://www.webyog.com

- *phpMyAdmin.* Это популярный инструмент администрирования, который работает на веб-сервере и предоставляет веб-интерфейс к серверам MySQL. Хотя доступ к браузеру иногда бывает приятным, phpMyAdmin — это большой и сложный инструмент, и говорят о том, что у него много проблем с безопасностью. Будьте с ним предельно осторожны. Мы рекомендуем не устанавливать его на систему, доступную из Интернета. Более подробную информацию можно найти на странице http://sourceforge.net/projects/phpmyadmin/.
- *Adminer.* Это облегченный, безопасный инструмент администрирования на основе браузера, который относится к той же категории, что и phpMyAdmin. Разработчик позиционирует его как наилучшую замену phpMyAdmin. Хотя он кажется более безопасным, мы по-прежнему рекомендуем соблюдать осторожность при установке его в любом общедоступном месте. Дополнительная информация доступна по адресу http://www.adminer.org.

Утилиты командной строки

MySQL поставляется с некоторыми утилитами командной строки, такими как mysqladmin и mysql-check. Они перечислены и задокументированы в руководстве по MySQL. Сообщество, сложившееся вокруг MySQL, в дополнение к этим утилитам создало широкий набор высококачественных инструментов с хорошей документацией.

- *Percona Toolkit.* Percona Toolkit — обязательный набор инструментов для администраторов MySQL. Он является наследником Maatkit и Aspersa — наборов инструментов, ранее созданных Бэроном, которые множество пользователей называют обязательными для всех, кто серьезно развертывает MySQL. Он включает в себя множество инструментов для таких целей, как анализ журналов, проверка целостности репликации, синхронизация данных, анализ схем и индексирования, консультирование по запросам и архивирование данных. Если вы только начинаете работать с MySQL, предлагаем сначала изучить следующие основные инструменты: pt-mysql-summary, pt-table-checksum, pt-table-sync и pt-query-digest. Дополнительная информация доступна по адресу http://www.percona.com/software/.
- *Maatkit и Aspersa.* Эти два инструмента в той или иной форме существуют с 2006 года, и оба считались полезными для пользователей MySQL. Сейчас они объединены в Percona Toolkit.
- *Набор openark.* Этот набор, созданный Шломи Ноачем (Shlomi Noach) (http://code.openark.org/forge/openark-kit), содержит скрипты, написанные на языке Python, которые можно использовать для самых разных административных задач.
- *Утилиты MySQL Workbench.* Некоторые из утилит MySQL Workbench можно использовать в качестве обособленных скриптов Python. Их можно скачать на https://launchpad.net/mysql-utilities.

Помимо этих инструментов, существует множество других, которые разработаны и поддерживаются менее формально. Многие из известных членов сообщества,

сложившегося вокруг MySQL, время от времени создавали свои инструменты, в основном размещая их на своих сайтах или на MySQL Forge (http://forge.mysql.com). Разнообразную информацию можно найти в блоге Planet MySQL (http://planet.mysql.com), но, к сожалению, для всех этих инструментов нет единого централизованного каталога.

Утилиты SQL

Существует множество бесплатных надстроек и утилит, которые можно использовать на самом сервере, некоторые из них довольно мощные.

- *common_schema*. Проект Шломи Ноача common_schema (http://code.openark.org/forge/common_schema) — это мощный набор подпрограмм и представлений для серверных скриптов и администрирования. Для MySQL common_schema — как jQuery для JavaScript.
- *mysql-sr-lib*. Джузеппе Максиа (Giuseppe Maxia) создал библиотеку хранимых процедур для MySQL, которую вы можете найти по адресу http://www.nongnu.org/mysql-sr-lib/.
- *Репозиторий UDF для MySQL*. Роланд Бауман (Roland Bouman) организовал создание набора пользовательских функций для MySQL, который доступен по адресу http://www.mysqludf.org.
- *MySQL Forge*. В MySQL Forge (http://forge.mysql.com) вы найдете сотни программ, скриптов, фрагментов кода, утилит, подсказок и трюков, распространяемых сообществом.

Инструменты мониторинга

Мониторинг MySQL — тема, заслуживающая отдельной книги. Это большая и сложная задача, причем разные приложения зачастую предъявляют различные требования. Однако мы можем рассказать о некоторых наиболее полезных инструментах и ресурсах, посвященных этой тематике.

Слово «мониторинг» чрезвычайно многозначно. Многие употребляют его, подразумевая, что собеседник знает, о чем идет речь. Но наш опыт показывает, что в большинстве центров, где используется MySQL, имеется необходимость в различных видах мониторинга.

Мы уделим внимание инструментам как для интерактивного, так и для неинтерактивного мониторинга. Неинтерактивный мониторинг обычно подразумевает наличие автоматизированной системы, которая собирает результаты измерений и может уведомить администратора о том, что какой-то параметр вышел за пределы безопасного диапазона. Инструменты интерактивного мониторинга позволяют наблюдать за работой сервера в режиме реального времени. Каждая категория будет далее описана в отдельном разделе.

Вас могут заинтересовать и другие различия между инструментами, например, для пассивного (примером может служить innotop) и активного мониторинга. В последнем случае инструмент может посылать оповещения или инициировать некоторые действия (таков, например, Nagios). А быть может, вы ищете утилиту, которая создает хранилище информации, а не просто отображает текущую статистику. По ходу изложения будем отмечать и такие детали.

Наш опыт свидетельствует о том, что большинство вычислительных центров MySQL в первую очередь нуждаются в двух видах мониторинга: мониторинге работоспособности для обнаружения ситуаций, когда что-то идет не так, и предупреждении о них и записи показателей для трендов, диагностики, устранения неполадок, планирования мощности и т. д. Большинство систем хорошо справляются только с одной из этих задач и не могут справиться с обеими. К сожалению, существуют десятки инструментов, так что процесс выбора наиболее подходящего становится очень трудоемким.

Большинство систем мониторинга не проектировались специально для мониторинга серверов MySQL. Напротив, они представляют собой системы общего назначения, предназначенные для периодической проверки состояния многих видов ресурсов, от компьютеров и маршрутизаторов до программного обеспечения (например, MySQL). Они обычно имеют модульную архитектуру и часто поставляются как плагины для MySQL.

Как правило, система мониторинга устанавливается на своем собственном сервере и используется для мониторинга других серверов. Если вы используете ее для мониторинга важных систем, он быстро станет важной частью инфраструктуры, поэтому вам, скорее всего, придется предпринять дополнительные шаги, например сделать систему мониторинга избыточной с аварийным переключением.

Инструменты мониторинга с открытым исходным кодом

Приведем наиболее популярные системы мониторинга с открытым исходным кодом, решающие обе задачи.

- *Nagios.* Nagios (http://www.nagios.org) — вероятно, самая популярная система с открытым исходным кодом для обнаружения неполадок и предупреждения о них. Она периодически проверяет заданные службы и сравнивает результаты с определенными явно или по умолчанию лимитами. Если результаты выходят за пределы заданного диапазона, то Nagios может запустить программу и/или оповестить кого-то о неполадках. Система регистрации ответственных лиц и оповещений, реализованная в Nagios, позволяет поднимать тревогу, направляя оповещение другому лицу, изменять уведомления или отправлять их в разные места в зависимости от времени суток и других условий, а также учитывать запланированное время вывода системы из эксплуатации. Nagios также понимает, каковы зависимости между службами, поэтому не будет беспокоить вас сообщением об остановке MySQL, если замечает, что сервер недоступен из-за того, что вышел

из строя маршрутизатор на пути к нему или остановлен сам сервер, на котором работает MySQL.

Nagios может запускать любой исполняемый файл как подключаемый модуль при условии, что он принимает определенные параметры и генерирует ожидаемый результат. Поэтому дополнительные компоненты для Nagios пишутся на разных языках, включая язык оболочки, Perl, Python, Ruby и другие языки скриптов. И если не удается найти модуль, который делает именно то, что вам надо, то совсем несложно написать свой собственный. От него требуется лишь умение принимать стандартные аргументы, завершаться с определенным кодом состояния и, возможно, выводить текст, который Nagios сможет перехватить.

Однако у Nagios есть серьезные недостатки. Даже хорошо изучив эту систему, поддерживать ее довольно трудно. К тому же вся конфигурационная информация хранится в файлах, а не в базе данных. У этих файлов специальный синтаксис, в котором легко допустить ошибку, а модифицировать конфигурацию по мере развития и роста наблюдаемых систем — весьма трудоемкое занятие. Nagios не очень хорошо расширяется: вы можете легко писать плагины для мониторинга, но это все, что можно сделать. Наконец, средства построения графиков, анализа трендов и визуализации у нее ограничены. Nagios может хранить некоторые показатели производительности и другие сведения в базе данных MySQL и генерировать графики по этим данным, но все же в этом отношении она уступает в гибкости иным системам. Все эти проблемы усугубляются политикой. Nagios раздваивалась минимум дважды из-за реальных или вымышленных трудностей работы с кодом и персоналом. Версии называются Opsview (http://www.opsview.com) и Icinga (http://www.icinga.org). Многие предпочитают использовать именно эти системы, а не Nagios.

Системе Nagios посвящено несколько книг. Нам нравится книга Вольфганга Барта (Wolfgang Barth) *Nagios System and Network Monitoring* (издательство No Starch Press).

- *Zabbix*. Это полнофункциональная система мониторинга и сбора показателей. Вся конфигурационная информация и прочие данные хранятся в базе, а не в конфигурационных файлах. Кроме того, количество типов сохраняемых данных больше, чем в Nagios, поэтому она лучше анализирует тенденции и генерирует более качественные отчеты. Средства построения графиков и визуализации также значительно лучше, чем в Nagios, и многие считают, что ее проще конфигурировать, она более гибкая и лучше масштабируется. Дополнительную информацию можно найти на сайте http://www.zabbix.com.
- *Zenoss*. Программа написана на языке Python и имеет веб-интерфейс, построенный на основе технологии Ajax, что повысило быстродействие и продуктивность. Она умеет автоматически обнаруживать ресурсы в сети и объединяет в одном унифицированном инструменте средства мониторинга, оповещения, анализа тенденций, построения графиков и сохранения ранее полученных данных. Zenoss использует протокол SNMP для сбора данных с удаленных компьютеров, но может работать и по протоколу SSH. Имеется также поддержка подключаемых модулей Nagios. Дополнительную информацию можно найти на сайте http://www.zenoss.com.

- *Hyperic HQ.* Это система мониторинга, написанная на языке Java, она в большей степени ориентирована на так называемый корпоративный мониторинг, чем другие системы этого класса. Как и Zenoss, умеет автоматически обнаруживать ресурсы и поддерживает подключаемые модули Nagios, но архитектурно устроена иначе и более тяжеловесна. Дополнительную информацию можно найти на сайте http://www.hyperic.com.
- *OpenNMS.* Система написана на Java, вокруг нее сложилось активное сообщество разработчиков. Она обладает обычными средствами, такими как мониторинг и оповещение, к которым добавляет построение графиков и анализ тенденций. При проектировании ставились следующие цели: высокая производительность и масштабируемость, автоматизация и гибкость. Как и Hyperic, она предназначена для мониторинга крупных критически важных систем масштаба предприятия. Дополнительную информацию можно найти на сайте http://www.opennms.org.
- *Groundwork Open Source.* Система построена на базе Nagios и объединяет Nagios с несколькими другими инструментами, предоставляя портальный интерфейс. Быть может, правильнее всего описать ее как систему, которую можно было бы построить самостоятельно, будь вы экспертом в программах Nagios, Cacti и целом ряде других и располагай временем для их интеграции друг с другом. Дополнительную информацию можно найти на сайте http://www.groundworkopensource.com.

Помимо систем, решающих обе задачи, существует множество программ, которые сосредоточены на сборе показателей и позволяют представлять их графически и визуализировать, но при этом не проверяют работоспособность. Многие из них построены поверх RRDTool (http://www.rrdtool.org), который хранит данные временных рядов в файлах с циклической базой данных (RRD). Они агрегируют поступающие данные, интерполируют отсутствующие значения, если входные сведения не поступают в ожидаемое время, и располагают мощным инструментарием для генерации красивых графиков. На базе RRDTool разработаны несколько систем. Вот самые популярные.

- *MRTG.* Программа Multi Router Traffic Grapher (http://oss.oetiker.ch/mrtg/) представляет собой квинтэссенцию систем на базе RRDTool. Она предназначена для протоколирования сетевого трафика, но может быть обобщена для протоколирования и визуализации других данных.
- *Cacti.* Cacti (http://www.cacti.net) — возможно, самая популярная система на базе RRDTool. Это PHP-интерфейс к RRDTool. Она использует базу данных MySQL для определения серверов, плагинов, графиков и т. д. Системой управляют шаблоны, поэтому вы можете определить их, а затем применить к своим системам. Бэрон написал очень популярный набор шаблонов для MySQL и других систем (дополнительную информацию можно получить на http://code.google.com/p/mysql-cacti-templates/). Они были перенесены в Munin, OpenNMS и Zabbix.
- *Ganglia.* Система Ganglia (http://ganglia.sourceforge.net) аналогична Cacti, но предназначена для мониторинга кластеров и grid-систем, поэтому позволяет просматривать обобщенные сведения по нескольким серверам с возможностью детализировать отчет до уровня отдельного сервера.

- *Munin.* Система Munin (http://munin.projects.linpro.no) собирает данные, передает их RRDTool, а затем генерирует по ним графики с различными уровнями детализации. На основе конфигурационных параметров она создает статические HTML-файлы, которые можно посмотреть в браузере и легко выявить тенденции. Определить график нетрудно — требуется создать подключаемый скрипт, который выводит информацию в специальном синтаксисе. Munin интерпретирует эти данные как инструкции по построению графика.

Системы на основе RRDTool имеют некоторые ограничения, такие как невозможность запросить сохраненные данные с помощью стандартного языка запросов, невозможность бессрочно хранить данные, проблемы с видами данных, которые не вписываются легко в простые счетчики или датчики, требование предопределять показатели и графики и т. д. В идеале мы хотели бы иметь систему, которая может просто принимать любые показатели, которые вы ей отправляете, без предварительного определения того, что они собой представляют, и впоследствии строить на их основе произвольные графики, также без необходимости предопределять их. Вероятно, из того, что мы видели, ближе всего к этому идеалу система Graphite (http://graphite.wikidot.com).

Эти системы могут использоваться для сбора, записи и отображения данных и отчетов в системах MySQL с разной степенью гибкости и различающимися целями. Но им не хватает действительно гибкого средства оповещения, когда что-то идет не так.

Основная проблема большинства упомянутых нами систем заключается в том, что они были специально разработаны людьми, недовольными тем, что существующие системы не вполне соответствуют их потребностям. Поэтому они написали еще одну систему, которая не вполне соответствует множеству других потребностей других людей. У большинства этих систем есть фундаментальные ограничения, такие как странная внутренняя модель данных, которая не всегда хорошо работает. Неприятно об этом говорить, но в большинстве случаев использование одной из этих систем похоже на попытку забить круглый колышек в квадратное отверстие.

Коммерческие системы мониторинга

Хотя мы знаем множество пользователей MySQL, которые любят использовать инструменты с открытым исходным кодом, но знаем и тех, которые готовы платить за патентованное программное обеспечение, если оно улучшает работу и экономит время и усилия. Вот некоторые из доступных коммерческих вариантов.

- *MySQL Enterprise Monitor.* Этот инструмент включен в подписку на поддержку MySQL от Oracle. Он сочетает в себе такие возможности, как мониторинг, измерение показателей и построение графиков, консультационные услуги и анализ запросов. Он использует агента на серверах для мониторинга своих счетчиков состояния, включая ключевые показатели операционной системы. Он может фиксировать запросы двумя способами: через MySQL Proxy или с помощью соответствующих соединителей MySQL, таких как Connector/J для Java или MySQLi для PHP. Хотя инструмент спроектирован для мониторинга MySQL, его можно

несколько расширить. Тем не менее вы вряд ли найдете его адекватным средством мониторинга всех серверов и служб в своей инфраструктуре. Дополнительную информацию можно получить по адресу http://www.mysql.com/products/en terprise/monitor.html.

- *MONyog.* MONyog (http://www.webyog.com) — это система мониторинга без агентов, основанная на браузере и предназначенная для работы на пользовательском компьютере. Она запускает HTTP-сервер. Вы можете указать этот сервер браузеру и использовать систему.
- *New Relic.* New Relic (http://newrelic.com) представляет собой размещенную в Интернете и поставляемую как услугу систему управления производительностью приложений, которая может анализировать производительность всего вашего приложения, начиная с кода приложения (на Ruby, PHP, Java и других языках) и заканчивая JavaScript, выполняемым на браузере, вызовами SQL, которые вы делаете к базе данных, пространством на диске сервера, использованием процессора и другими показателями.
- *Circonus.* Circonus (https://circonus.com) представляет собой размещенную в Интернете систему мониторинга SaaS и систему оповещения от OmniTI. Агент собирает показатели с одного или нескольких серверов и пересылает их в Circonus, где вы просматриваете их через панель управления на основе браузера.
- *Monitis.* Monitis (http://monitis.com) — это еще одна система мониторинга SaaS с облачным хостингом. Она предназначена для мониторинга всего, а это означает, что она ничем особо не примечательна. У нее есть бесплатный кузен с минимальной конфигурацией Monitor.us (http://mon.itor.us), у которого также есть плагин для MySQL.
- *Splunk.* Splunk (http://www.splunk.com) — это агрегатор журналов и поисковая система, которые помогут вам получить оперативное представление обо всех данных, сгенерированных компьютером в вашей среде.
- *Pingdom.* Pingdom (http://www.pingdom.com) удаленно контролирует доступность и производительность вашего сайта. Существует множество таких сервисов, как Pingdom, и мы не готовы рекомендовать какой-то конкретный, однако действительно советуем использовать какую-либо внешнюю службу мониторинга, которая сможет известить вас, если ваш сайт недоступен. Многие службы могут делать гораздо больше, чем просто пинговать или скачивать веб-страницу.

Есть множество других инструментов коммерческого мониторинга — мы могли бы по памяти назвать десяток или больше. Однако, используя любую систему мониторинга, необходимо обращать внимание на их влияние на сервер. Некоторые инструменты довольно неудачные, поскольку разработаны компаниями, не имеющими практического опыта работы с большими и сильно загруженными MySQL-системами. Например, мы решили не одну аварийную проблему отключением функции системы мониторинга, которая выполняла `SHOW TABLE STATUS` в каждой базе данных каждую минуту. (Эта команда чрезвычайно разрушительна для больших систем с большой нагрузкой на операции ввода/вывода.) Инструменты, которые слишком часто запрашивают некоторые таблицы `INFORMATION_SCHEMA`, также могут иметь негативные последствия.

Мониторинг из командной строки с использованием Innotop

Создано несколько инструментов мониторинга на основе командной строки, большинство из которых тем или иным образом эмулируют инструмент top Unix. Самым сложным и функциональным является innotop (http://code.google.com/p/innotop/), который мы рассмотрим подробнее. Однако существуют и другие, например mtop (http://mtop.sourceforge.net), mytop (http://jeremy.zawodny.com/mysql/mytop/), а также клоны mytop с веб-интерфейсом.

Хотя mytop — это клон оригинального top для MySQL, innotop способен выполнять все, что может и он, а также многое другое, поэтому мы сфокусируемся именно на innotop.

Программу innotop написал Бэрон Шварц, один из авторов этой книги. Она в реальном времени дает представление о том, что происходит на вашем сервере. Несмотря на свое название, она не ограничивается только мониторингом InnoDB, а способна мониторить практически любой аспект MySQL. Она предоставляет возможность вести наблюдение сразу за несколькими экземплярами MySQL. Ко всему прочему, инструмент очень гибко конфигурируется и допускает расширение.

Перечислим некоторые возможности innotop:

- составление списка всех текущих транзакций InnoDB;
- создание списка выполняемых в текущий момент запросов;
- составление списка текущих блокировок и ожиданий блокировок;
- сбор сводной информации о состоянии сервера и переменных, отражающей относительные величины значений;
- установление режимов для вывода информации о внутреннем состоянии InnoDB, в частности о буферах, взаимоблокировках, ошибках внешнего ключа, вводе/выводе, операциях со строками, семафорах и пр.;
- мониторинг репликации, причем состояние главного и подчиненных серверов можно увидеть одновременно;
- задание режима для просмотра любых серверных переменных;
- реализация функции группировки серверов, которая позволяет удобно организовать их логическое расположение на экране;
- задание неинтерактивного режима для применения в командных скриптах.

Установить innotop нетрудно. Можно сделать это из репозитория пакетов для вашей операционной системы. Или загрузить исходный код с сайта http://code.google.com/p/innotop/, распаковать его и запустить стандартную процедуру `make install`:

```
perl Makefile.PL
make install
```

Установив программу, вызовите innotop из командной строки, и она проведет вас по всему процессу подключения к серверу MySQL. Поскольку innotop умеет читать конфигурационный файл `~/.my.cnf`, то, возможно, вам придется указать лишь имя компьютера, на котором работает сервер, и несколько раз нажать клавишу Enter.

После подключения вы окажетесь в режиме T (InnoDB Transaction) и увидите список текущих транзакций InnoDB, показанный на рис. 16.1

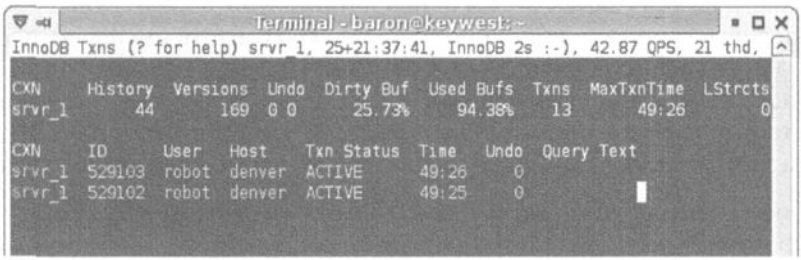


Рис. 16.1. innotop в режиме T (Transaction)

По умолчанию innotop применяет фильтры, чтобы не загромождать экран (как и везде в innotop, вы можете писать собственные или модифицировать встроенные фильтры). На рис. 16.1 большинство транзакций отфильтровано, оставлены только пять активных. Чтобы отключить фильтр и вывести на экран столько транзакций, сколько поместится, нажмите клавишу I.

В этом режиме innotop отображает заголовок и главный список потоков. В заголовке показана сводная информация об InnoDB, например длина списка истории, количество неудаленных транзакций, процентная доля «грязных» буферов в пуле и т. п.

Первым делом нажмите клавишу ?, чтобы просмотреть справку. Содержимое этого окна зависит от того, в каком режиме работает innotop, но в любом случае будет приведен список горячих клавиш, чтобы можно было увидеть все доступные действия. На рис. 16.2 изображено окно справки в режиме T.

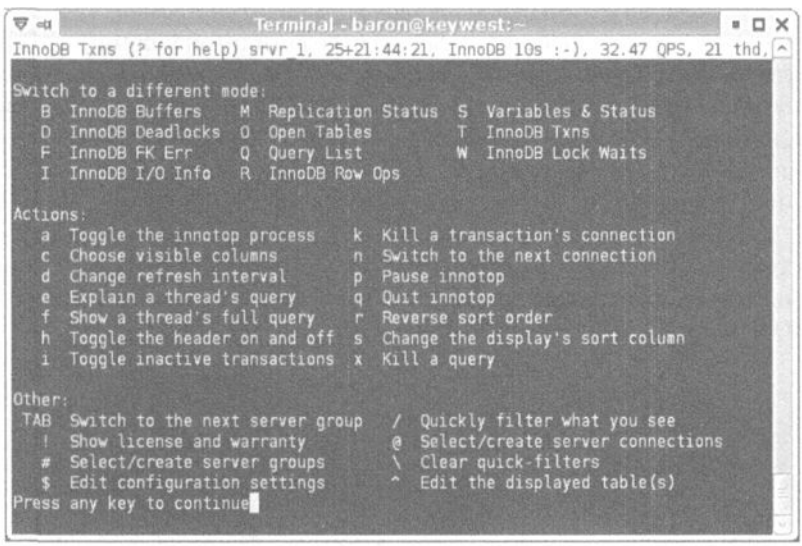


Рис. 16.2. Окно справки innotop

Мы не станем подробно рассматривать остальные режимы, но, как легко понять, глядя на окно справки, возможности innotop весьма обширны.

Единственное, на чем еще мы хотим остановиться, — это выполнение простенькой настройки с целью продемонстрировать, как можно следить именно за тем, что вы хотите. Одна из сильных сторон innotop — умение интерпретировать определенные пользователем выражения, например, выражение `Uptime/Questions` обозначает параметр «количество запросов в секунду». Можно вывести результат, рассчитанный за время с момента запуска сервера и/или инкрементно с момента последнего опроса.

Таким способом можно добавлять собственные столбцы в отображаемые таблицы. Например, в режиме Q (Query List — список запросов) имеется заголовок, в котором демонстрируется сводная информация о сервере. Посмотрим, как модифицировать его, чтобы отображались сведения о заполненности кэша ключей. Запустите innotop и нажмите клавишу Q для входа в режим Q. Результат будет выглядеть так, как показано на рис. 16.3.

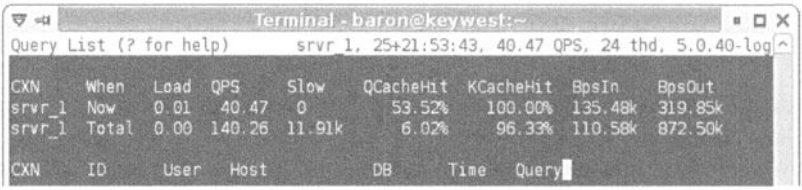


Рис. 16.3. innotop в режиме Q (Query List)

Этот снимок экрана обрезан, поскольку в данном случае сам список запросов нас не интересует — важен только заголовок.

В заголовке показана статистика «Сейчас» (измеряет инкрементную активность с момента последнего извлечения данных с сервера утилитой innotop) и «Всего» (измеряет активность с момента запуска сервера MySQL 25 дней назад). Каждое поле в заголовке является результатом вычисления некоторого выражения, содержащего значения, которые возвращают команды `SHOW STATUS` и `SHOW VARIABLES`. Отображаемые по умолчанию поля (см. рис. 16.3) встроены, но легко добавить и собственные. Нужно лишь включить соответствующий столбец в таблицу заголовка. Нажмите клавишу ^, чтобы запустить редактор таблиц, а затем наберите `q_header` в ответ на вопрос о том, какой заголовок вы собираетесь изменять (рис. 16.4). Механизм завершения, вызываемый нажатием клавиши Tab, встроен, так что достаточно нажать клавишу Q, а затем Tab — для завершения слова.

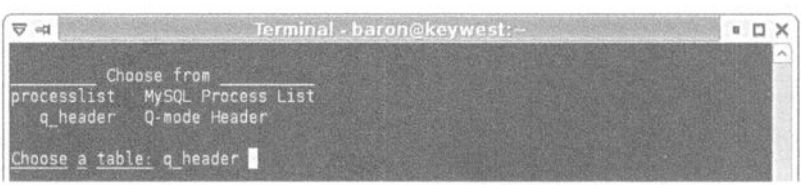


Рис. 16.4. Добавление заголовка (начало)

После этого вы увидите определение заголовка в режиме Q (рис. 16.5). Оно представлено в виде таблицы, в которой выделен первый столбец. Мы можем перемещать выделение, переупорядочивать и редактировать столбцы, а также выполнять другие действия (чтобы увидеть полный перечень, нажмите клавишу ?). Но сейчас мы хотим просто создать новый столбец. Нажмите клавишу N и введите имя столбца (рис. 16.6).

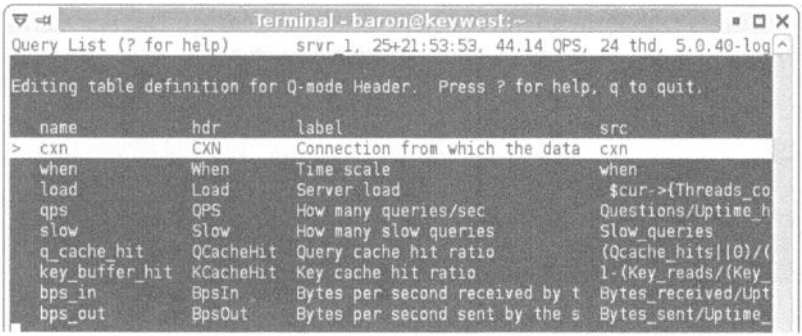


Рис. 16.5. Добавление заголовка (выбор)

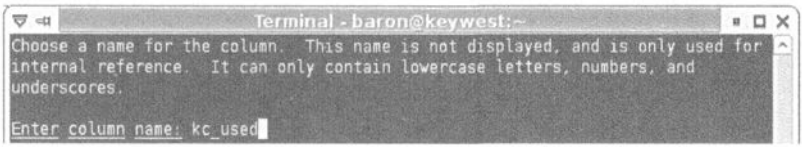


Рис. 16.6. Добавление заголовка (присвоение имени столбцу)

Затем введите заголовок столбца (рис. 16.7) и выберите источник данных для него. Это выражение, которое innotop компилирует во внутреннее представление в виде функции. В выражении можно использовать имена переменных, возвращаемых командами `SHOW VARIABLES` и `SHOW STATUS`. Мы добавили скобки и Perl-оператор `or`, чтобы предотвратить деление на нуль, но в остальном это выражение не содержит ничего сложного. Мы также воспользовались встроенной в innotop функцией преобразования `percent()`, чтобы отформатировать результат в виде процента (дополнительную информацию см. в документации по innotop). Выражение показано на рис. 16.8.

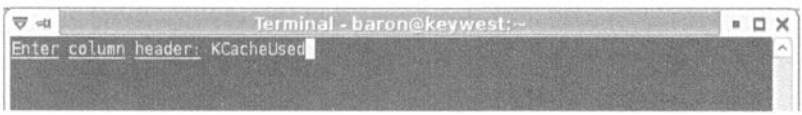


Рис. 16.7. Добавление заголовка (текста над столбцом)

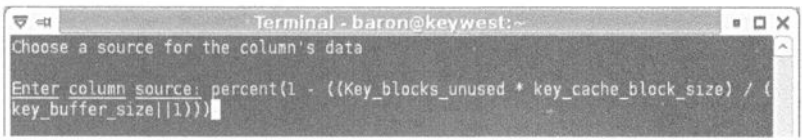


Рис. 16.8. Добавление заголовка (вычисляемое выражение)

Нажав Enter, вы увидите то же определение таблицы, что и прежде, но внизу появится описание добавленного столбца. Несколько раз нажмите клавишу +, чтобы поднять этот столбец выше и разместить его в списке после столбца `key_buffer_hit`, а затем Q, чтобы выйти из редактора. Вуаля — новый столбец расположился между `KCacheHit` и `BpsIn` (рис. 16.9). Как видите, настроить программу innotop так, чтобы она отслеживала именно то, что вам нужно, совсем не трудно. Можно даже написать подключаемый модуль, если по-другому ничего не получается. Полная документация по innotop имеется на сайте http://code.google.com/p/innotop/.

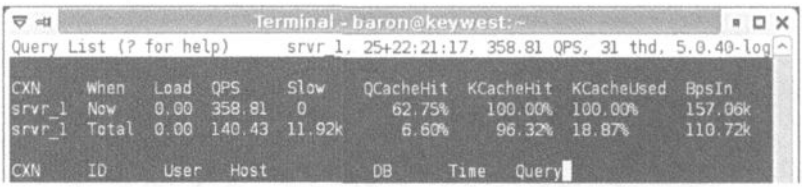


Рис. 16.9. Добавление заголовка (результат)

Итоги главы

Для администрирования MySQL необходимы хорошие инструменты. Вы получили полезные советы о применении некоторых превосходных инструментов: они доступны, широко протестированы и популярны, как, например Percona Toolkit (в девичестве Maatkit). Приступая к работе на новом сервере, самое первое, что мы делаем, — запускаем pt-summary и pt-mysql-summary. Когда мы работаем на сервере, то, как правило, наблюдаем за ним и связанными с ним серверами на другом терминале с помощью утилиты innotop.

Инструменты мониторинга являются более сложной темой, поскольку они очень важны для организации. Если вы сторонник программ с открытым исходным кодом и хотите использовать такие системы мониторинга, можете попробовать либо комбинацию Nagios и Cacti с шаблонами для Cacti, созданными Бэроном, либо Zabbix, если вас не смущает его сложный интерфейс. Если собираетесь применить для мониторинга MySQL коммерческий инструмент, можете выбрать MySQL Enterprise Monitor, который неплохо работает у многих знакомых нам пользователей. Если же хотите иметь что-то, способное контролировать всю вашу среду и все программное обеспечение и оборудование в ней, вам нужно будет провести собственное исследование — это более значительная тема, чем мы хотели затронуть в этой книге.

Приложения

Приложение А. Ветки и версии MySQL

В главе 1 мы рассмотрели историю поглощения MySQL компанией Sun Microsystems, а затем приобретения Sun корпорацией Oracle и то, как сервер пережил эти изменения в управлении. Но у этой истории есть продолжение. MySQL теперь доступна не только для Oracle. В процессе двух поглощений появилось несколько вариантов MySQL. Хотя большинство пользователей вряд ли захотят использовать что-либо отличное от официальной версии MySQL от Oracle, варианты на самом деле важны и оказали существенное влияние на всех пользователей MySQL, даже тех, кто никогда и не подумал бы об их использовании.

За эти годы было создано несколько вариантов MySQL, но испытание временем выдержали только три основных — Percona Server, MariaDB и Drizzle. Вокруг каждого из них сформировалось активное сообщество пользователей, а кроме того, у них есть и некоторая коммерческая поддержка. Их все поддерживают независимые поставщики услуг.

Как создатели Percona Server, мы в некоторой степени предвзяты, однако считаем, что это приложение здесь вполне уместно, поскольку мы обеспечиваем обслуживание, поддержку, консультирование, обучение и разработку для всех вариантов MySQL. Кроме того, мы пригласили Брайана Акера (Brian Aker) и Монти Видениуса (Monty Widenius), которые создали Drizzle и MariaDB соответственно, для участия в написании этого приложения, так что это будет не только наша версия истории.

Percona Server

Percona Server (http://www.percona.com/software/) вырос из наших усилий, направленных на решение проблем клиентов. Во втором издании этой книги упоминались некоторые из исправлений, которые мы сделали для улучшения журналирования сервера MySQL и его оснащения инструментами. Это было действительно зарождение Percona Server. Мы изменили исходный код сервера, столкнувшись с проблемами, которые нельзя было решить другим способом.

Percona Server должен обеспечивать достижение трех основных целей.

- *Прозрачность*. Добавленные инструменты позволяют пользователям лучше исследовать внутренние элементы и поведение сервера. Сюда входят такие функции, как счетчики в `SHOW STATUS`, таблицы в `INFORMATION_SCHEMA` и особенно новый уровень детализации в журнале медленных запросов.
- *Производительность*. Percona Server включает в себя множество улучшений производительности и масштабируемости. Производительность важна сама по себе,

однако Percona Server повышает также предсказуемость и стабильность уровня производительности. Основной акцент сделан на подсистеме хранения InnoDB.

- *Операционная гибкость.* Percona Server содержит множество возможностей, устраняющих ограничения. Хотя некоторые ограничения кажутся несерьезными, тем не менее они могут затруднить для оперативного персонала и системных администраторов запуск MySQL в качестве надежного и стабильного компонента их инфраструктуры.

Percona Server является обратно совместимой упрощенной заменой для MySQL с минимальными изменениями, не изменяющими синтаксис SQL, клиент-серверный протокол или форматы файлов на диске[1]. Все, что работает на MySQL, будет работать без изменений и на Percona Server. Для перехода на Percona Server требуется только выключить MySQL и запустить Percona Server, экспортировать и повторно импортировать данные не нужно. Обратное переключение также безболезненно, и это на самом деле очень важно: множество проблем можно решить, если выполнить временное переключение, использовать для диагностики проблемы усовершенствованный инструментарий, а затем возвратиться к стандартной MySQL.

Мы выбираем улучшения, которые отклоняются от стандартной MySQL только там, где это необходимо и приносит значительную выгоду. Мы полагаем, что большинству пользователей вполне хватает возможностей официальной версии MySQL, распространяемой Oracle, и стараемся не уходить от нее далеко.

Percona Server включает в себя подсистему хранения Percona XtraDB, расширенную версию InnoDB. В ней также реализована обратная совместимость. Например, если вы создаете таблицу в подсистеме хранения InnoDB, Percona Server автоматически распознает ее и будет использовать Percona XtraDB. Percona XtraDB также входит в состав MariaDB.

Некоторые из усовершенствований Percona Server были включены в версию MySQL Oracle, а ряд других — реализованы несколько иным способом. В результате Percona Server стал своего рода предварительной демонстрацией возможностей, которые в отдельных случаях позже могут появиться в стандартной MySQL. Многие из улучшений, сделанных в Percona Server 5.1 и 5.5, вероятно, будут реализованы в MySQL 5.6.

Maria DB

После того как Sun приобрела MySQL, Монти Видениус, один из создателей MySQL, покинул Sun Microsystems из-за разногласий, возникших в ходе работы над MySQL. Он основал Monty Program AB и создал MariaDB для культивирования «открытой среды разработки, которая поощряла бы внешнее участие». Цели MariaDB — это развитие сообщества, исправление ошибок, реализация множества новых возможностей, а также интеграция разработанных участниками сообщества функций. Процитируем

[1] Исторически сложилось так, что в форматы файлов было внесено несколько изменений. По умолчанию они отключены, но при желании их можно включить.

Монти еще раз[1]: «Концепция MariaDB заключается в ориентации на пользователя и клиента, а также на широкое использование патчей и плагинов, разработанных членами сообщества».

Каковы отличительные особенности MariaDB? По сравнению с Percona Server она включает в себя гораздо более широкие изменения на сервере. (Большинство крупных новшеств в Percona Server реализованы в подсистеме хранения Percona XtraDB, а не на уровне сервера.) Например, реализовано много изменений в оптимизаторе запросов и репликации. Кроме того, она использует подсистему хранения Aria вместо MyISAM для внутренних временных таблиц (тех, которые используются для сложных запросов, таких как запросы с ключевым словом `DISTINCT` или подзапросы). Aria первоначально называлась Maria и во времена Sun рассматривалась как замена InnoDB. По сути, это отказоустойчивая версия MyISAM.

MariaDB, помимо Percona XtraDB и Aria, содержит ряд подсистем хранения, разработанных членами сообщества, например SphinxSE и PBXT.

MariaDB является дополнением к существующей MySQL, поэтому все приложения должны работать без изменений, как и в случае с Percona Server. Тем не менее она будет работать намного лучше при реализации некоторых сценариев, таких как сложные подзапросы или соединения многих таблиц. Кроме того, в ней реализован сегментированный кэш ключей MyISAM, что делает MyISAM гораздо более масштабируемой на современном оборудовании.

Возможно, самым лучшим вариантом MariaDB является MariaDB 5.3 — на момент написания книги существовала ее предвыпускная версия. Во время подготовки этой версии была проведена огромная работа над оптимизатором запросов — сделаны, возможно, самые большие улучшения оптимизатора, которые MySQL видела за десятилетие. В ней добавлены новые планы выполнения запросов, такие как хеш-соединения, и исправлены многие проблемные моменты, на которые мы указали в этой книге, например внешнее выполнение подзапроса. Кроме того, реализованы значительные расширения для сервера, такие как динамические столбцы, управление доступом на основе ролей и поддержка микросекундной метки времени.

Для того чтобы познакомиться с более полным списком улучшений, сделанных в MariaDB, прочитайте документацию на сайте http://www.askmonty.org или на следующих сайтах (здесь сведены все изменения): http://askmonty.org/blog/the-2-year-old-mariadb/ и http://kb.askmonty.org/en/what-is-ma riadb-53.

Drizzle

Drizzle — это настоящая ветка MySQL, а не просто вариант или усовершенствование. Он несовместим с MySQL, хотя и не настолько отличается от нее, чтобы быть не-

[1] Цитаты взяты с сайтов http://monty-says.blogspot.com/2009/02/time-to-move-on.html и http://monty-says.blogspot.com/2010/03/time-flies-one-year-of-mariadb.html.

узнаваемым. В большинстве случаев вы не можете просто отключить сервер MySQL и заменить его на Drizzle из-за сделанных в нем изменений, таких как необычный синтаксис SQL.

Drizzle был создан в 2008 году, чтобы лучше удовлетворять потребности пользователей MySQL. Он разработан для реализации основных функций, необходимых веб-приложениям. Он четко организован и значительно упрощен по сравнению с MySQL, количество вариантов в нем гораздо меньше — например, для хранения символов используется только формат UTF8 и существует лишь один тип `BLOB`. Он построен в основном для 64-битного оборудования и поддерживает IPv6-сеть.

Одной из ключевых задач сервера базы данных Drizzle является исключение существующих в MySQL неожиданностей поведения и устаревшего функционала, таких как объявление столбца `NOT NULL`, а затем обнаружение того, что база данных каким-то образом хранит в ней значение `NULL`. Плохо реализованные или громоздкие функции, которые можно обнаружить в MySQL, такие как триггеры, кэш запросов и команда `INSERT ON DUPLICATE KEY UPDATE`, просто удалены.

Архитектура Drizzle представляет собой микроядра с гибким ядром и плагинами. Ядро сервера разделено на гораздо меньшие фрагменты кода, чем MySQL. Почти все функции являются подключаемыми, даже такие как `SLEEP()`. Это позволяет очень просто и продуктивно работать с Dizzle на уровне исходного кода.

Drizzle использует стандартные библиотеки с открытым исходным кодом, такие как Boost, и соответствует стандартам в кодировании, построении инфраструктуры и API. Она применяет Google Protocol Buffers — формат открытых сообщений для таких целей, как репликация, и использует модифицированную версию InnoDB в качестве подсистемы хранения по умолчанию.

Специалисты Drizzle очень рано начали эталонное тестирование на сервере с использованием стандартных эталонных тестов с точностью до 1024 потоков для измерения производительности при высокой конкурентности. Повышение производительности при высоком уровне конкурентности важнее, чем при низком, и в этом смысле были достигнуты немалые успехи.

Drizzle создается сообществом пользователей, и их вклад в его разработку как программы с открытым исходным кодом намного больше, чем в MySQL. Лицензия сервера — это чистая GPL без двойного лицензирования. Однако — и это один из самых важных аспектов для разработки коммерческой экосистемы — имеется новая клиентская библиотека, которая использует клиент-серверный протокол MySQL, но лицензируется по принципу, действующему в BSD. Это означает, что вы можете создать запатентованное приложение, которое подключается к MySQL через клиентскую библиотеку Drizzle, и вам не нужно приобретать коммерческую лицензию в клиентской библиотеке MySQL или делать свое программное обеспечение доступным по лицензии GPL. Существование клиентской библиотеки libmysql для MySQL было одной из основных причин того, что компании покупали коммерческие лицензии для MySQL — без коммерческой лицензии, которая позволяла им ссылаться на libmysql, они были бы вынуждены выпускать свое программное

обеспечение под GPL. Этого больше не требуется, поскольку теперь они могут использовать библиотеку Drizzle.

Drizzle развертывается в некоторых производственных средах, но, насколько мы знаем, не очень широко. Философия проекта Drizzle, состоящая в отказе от обратной совместимости, означает, что эта база данных, скорее всего, должна использоваться для новых приложений, а не для переноса существующих.

Другие варианты MySQL

Имеются или по крайней мере существовали ранее множество других вариантов сервера MySQL. Крупные компании, такие как Google, Facebook и eBay, поддерживают измененные версии сервера, которые соответствуют их конкретным потребностям и сценариям развертывания. Большая часть исходного кода этих версий доступна широкой публике, возможно, наиболее известными примерами являются патчи для MySQL от Facebook и Google.

Кроме того, было несколько ветвей или более новых дистрибутивов, таких как OurDelta, DorsalSource и просуществовавший некоторое время дистрибутив от Хенрика Инго (Henrik Ingo).

Вдобавок многие не понимают, что, устанавливая MySQL из репозиториев пакетов дистрибутива GNU/Linux, они на самом деле получают модифицированную — в некоторых случаях довольно сильно — версию сервера. Red Hat и Debian (а значит, Fedora и Ubuntu) поставляют нестандартную версию MySQL, то же самое касается Gentoo и практически всех прочих дистрибутивов GNU/Linux. В отличие от других вариантов, которые мы назвали, эти дистрибутивы не сообщают, как сильно в них изменен исходный код сервера, поскольку в имена всех входит название MySQL.

В прошлом у нас было много проблем с модифицированными версиями MySQL. Это одна из причин, по которой мы склонны выступать за использование версии MySQL Oracle, если нет веских оснований этого не делать.

Итоги

Ветви и варианты MySQL редко содержали значительное количество кода, который включали в исходное кодовое дерево MySQL, но тем не менее они существенно повлияли на направление и темпы развития MySQL. В некоторых случаях они обеспечивают превосходную альтернативу. Должны ли вы использовать fork вместо официального MySQL Oracle? Мы не думаем, что в большинстве случаев это стоит делать. Выбор обычно основан на ощущениях (они никогда не бывают четкими) или бизнес-соображениях, таких как наличие общеорганизационных отношений с Oracle. Есть две категории людей, которые склонны отказаться от официальной версии сервера:

- те, кто столкнулся с конкретной проблемой, которую нельзя решить без изменения исходного кода;

- те, кто не доверяют руководству Oracle в отношении MySQL[1] и чувствуют себя более счастливыми с версией, которая, на их взгляд, является версией с действительно открытым исходным кодом.

Зачем вам выбирать какую-то конкретную ветвь? Мы потдытожим аргументы следующим образом. Если вы хотите быть как можно ближе к официальной версии MySQL, но получить лучшую производительность, удобство и полезные функции, выберите Percona Server. Выбирайте MariaDB, если вам удобнее работать с большими изменениями на сервере или если вам нужен широкий набор разработок членов сообщества, таких как, например, дополнительные механизмы хранения. Выберите Drizzle, если вам нужен скудный, урезанный сервер базы данных и вас не смущает то, что он несовместим с MySQL, однако вы хотите намного проще вносить свои собственные улучшения в базу данных.

Насколько популярны разные ветви и варианты? Никто толком не знает, но все согласны с тем, что, если объединить все развертывания неофициальных версий MySQL, они составят лишь небольшую долю от числа официальных развертываний MySQL в мире. Что касается относительной популярности, мы предвзяты, поскольку многие наши клиенты предпочитают использовать Percona Server, но, насколько мы видим, на практике самым популярным является Percona Server, на втором месте — MariaDB.

В целом все поставщики услуг имеют большой опыт работы с официальной MySQL, но, естественно, в Percona больше всего специалистов по работе с Percona Server, а сотрудникам Monty Program, соответственно, наиболее знакома MariaDB. Это очень важно, если вам нужны контракты на поддержку и исправление ошибок. Только Oracle может гарантировать, что ошибка будет исправлена в официальных версиях MySQL; другие поставщики могут вносить исправления, но не имеют возможности включать их в официальные релизы. Это один из ответов на вопрос о том, почему стоит выбирать ветвь: некоторые выбирают одну из ветвей просто потому, что это такая версия MySQL, которую их поставщик услуг полностью контролирует и может исправлять и улучшать.

[1] Как мы говорили в главе 1, мы на самом деле довольны, что Oracle является владельцем MySQL.

Приложение Б. Состояние сервера MySQL

На многие вопросы о сервере MySQL можно ответить, оценив его состояние. MySQL раскрывает информацию о своем внутреннем состоянии двумя способами: самый новый — база данных `PERFORMANCE_SCHEMA` в MySQL 5.5, однако стандартизированная база данных `INFORMATION_SCHEMA` существовала уже в версии MySQL 5.0, кроме того, имеется набор команд `SHOW`, которые были практически всегда. Кое-какая информация, которую можно получить с помощью команд `SHOW`, пока отсутствует в таблицах базы `INFORMATION_SCHEMA`.

Вы должны знать, как определить, что именно относится к вашей проблеме, как получить нужную информацию и как ее интерпретировать. Хотя MySQL предоставляет очень много сведений о том, что происходит внутри сервера, воспользоваться ими не так-то легко. Для понимания того, что есть что, нужно терпение, опыт и свободный доступ к руководству по MySQL. Кроме того, могут помочь хорошие инструменты.

Это приложение является главным образом справочным материалом, однако вы сможете найти здесь также кое-какую информацию о внутреннем устройстве сервера, особенно в разделах, посвященных InnoDB.

Системные переменные

MySQL показывает множество серверных системных переменных в результате применения команды `SHOW VARIABLES`. Их можно использовать в выражениях, а также в командной строке, запуская программу mysqladmin. Начиная с версии 5.1, доступ к ним можно получить через таблицы базы данных `INFORMATION_SCHEMA`.

Эти переменные представляют разнообразную конфигурационную информацию, например подразумеваемую по умолчанию подсистему хранения (`storage_engine`), поддерживаемые названия часовых поясов, схему упорядочения для соединения и параметры запуска. О том, как устанавливать и использовать системные переменные, мы рассказывали в главе 8.

Команда SHOW STATUS

Команда `SHOW STATUS` выводит переменные состояния сервера в виде таблицы с двумя столбцами, в которых содержатся имя и значение. В отличие от серверных переменных, упомянутых в предыдущем разделе, эти предназначены только для чтения. Посмотреть их можно с помощью SQL-команды `SHOW STATUS` или вызвав программу mysqladmin extended-status из командной строки. При использовании SQL-команды

можно использовать разделы `LIKE` и `WHERE` для фильтрации результатов: оператор `LIKE` выполняет стандартное сопоставление имени переменной с шаблоном. В обоих случаях возвращается таблица результатов, но ее нельзя сортировать, соединять с другими таблицами и выполнять иные стандартные операции, применимые к таблицам MySQL. В MySQL 5.1 и более поздних версиях можно извлекать значения непосредственно из таблиц `INFORMATION_SCHEMA.GLOBAL_STATUS` и `INFORMATION_SCHEMA.SESSION_STATUS`.

Мы используем термин «переменная состояния», говоря о переменных, которые выводит команда SHOW STATUS, и термин «серверная системная переменная», когда речь идет о конфигурационной переменной сервера.

Поведение команды `SHOW STATUS` в версии MySQL 5.0 существенно изменилось, но этого можно не заметить, если специально не присматриваться. Теперь MySQL поддерживает некоторые переменные глобально, а другие — на уровне отдельного соединения. Таким образом, `SHOW STATUS` выводит и глобальные, и сеансовые переменные. У многих из них двойная область видимости: глобальные и сеансовые переменные называются одинаково. Кроме того, команда `SHOW STATUS` теперь по умолчанию показывает только сеансовые переменные, так что, если вы привыкли просматривать с ее помощью глобальные, вас ждет разочарование. Для просмотра глобальных переменных нужно выполнить команду `SHOW GLOBAL STATUS`[1].

Существуют сотни переменных состояния, и с каждой новой версией их количество увеличивается. Большинство представляют собой счетчики либо содержат текущее значение какого-то показателя. Счетчики увеличиваются всякий раз, как MySQL выполняет какое-то действие, например инициирует полное сканирование таблицы (`Select_scan`). Показатели, в частности количество открытых соединений, могут как увеличиваться, так и уменьшаться. Иногда несколько переменных относятся к одному и тому же действию, например `Connections` (количество попыток соединения с сервером) и `Threads_connected` (количество подсоединенных потоков). В таком случае переменные взаимосвязаны, но из их имен это не всегда очевидно.

Счетчики хранятся в виде целых чисел без знака. В 32-разрядных сборках они занимают 4 байта, а в 64-разрядных — 8 байт. По достижении максимального значения счетчик обращаются в нуль. Когда ведется непрерывный мониторинг переменных, такой переход через нуль нужно отслеживать и учитывать: следует понимать, что если сервер проработал довольно долго, то наблюдаемое значение может оказаться меньше ожидаемого просто потому, что когда-то произошел переход через нуль (в 64-разрядных сборках эта проблема возникает гораздо реже).

Хороший способ оценить общую рабочую нагрузку состоит в сравнении значения внутри группы связанных переменных состояния, например при просмотре всех переменных вида `Select_*` или `Handler_*`. Если вы используете утилиту innotop, это легко сделать в режиме Command Summary, но можно и вручную с помощью команды

[1] Здесь вас подстерегает сюрприз: при использовании старой версии mysqladmin с новым сервером команда SHOW GLOBAL STATUS не вызывается, так что отображается «неправильная» информация.

`mysqladmin extended -r -i60 | grep Handler_`. Вот что показала утилита innotop для переменных вида `Select_*` на одном сервере из тех, что мы проверяли:

```
_____________________ Command Summary _____________________
Name                      Value   Pct     Last Incr  Pct
Select_scan              756582  59.89%          2  100.00%
Select_range             497675  39.40%          0    0.00%
Select_full_join           7847   0.62%          0    0.00%
Select_full_range_join     1159   0.09%          0    0.00%
Select_range_check            1   0.00%          0    0.00%
```

Первые два столбца отображают значения с момента загрузки сервера, а последние два — с момента последнего обновления (в данном случае это произошло 10 секунд назад). Процентные значения отражают долю от суммы значений, отображенных на экране, а не от суммы значений всех запросов.

Для сопоставления текущих и предыдущих снимков состояния и поиска различий между ними можно использовать также инструмент pt-mext из пакета Percona Toolkit или этот умный запрос созданный Шломи Ноачем[1]:

```
SELECT STRAIGHT_JOIN
    LOWER(gs0.VARIABLE_NAME) AS variable_name,
    gs0.VARIABLE_VALUE AS value_0,
    gs1.VARIABLE_VALUE AS value_1,
    (gs1.VARIABLE_VALUE - gs0.VARIABLE_VALUE) AS diff,
    (gs1.VARIABLE_VALUE - gs0.VARIABLE_VALUE) / 10 AS per_sec,
    (gs1.VARIABLE_VALUE - gs0.VARIABLE_VALUE) * 60 / 10 AS per_min
 FROM (
      SELECT VARIABLE_NAME, VARIABLE_VALUE
      FROM INFORMATION_SCHEMA.GLOBAL_STATUS
      UNION ALL
      SELECT '', SLEEP(10) FROM DUAL
    ) AS gs0
    JOIN INFORMATION_SCHEMA.GLOBAL_STATUS gs1 USING (VARIABLE_NAME)
 WHERE gs1.VARIABLE_VALUE <> gs0.VARIABLE_VALUE;
```

```
+----------------------+---------+---------+------+---------+---------+
| variable_name        | value_0 | value_1 | diff | per_sec | per_min |
+----------------------+---------+---------+------+---------+---------+
| handler_read_rnd_next | 2366    | 2953    |  587 |    58.7 |    3522 |
| handler_write        | 2340    | 3218    |  878 |    87.8 |    5268 |
| open_files           | 22      | 20      |   -2 |    -0.2 |     -12 |
| select_full_join     | 2       | 3       |    1 |     0.1 |       6 |
| select_scan          | 7       | 9       |    2 |     0.2 |      12 |
+----------------------+---------+---------+------+---------+---------+
```

Очень полезно посмотреть на значения всех этих переменных и показателей за последние несколько минут, а также за весь период работы сервера.

Далее приведен обзор — но не исчерпывающий список — различных категорий переменных, которые вы увидите в выводе команды `SHOW STATUS`. Для получения полной информации о данной переменной следует обратиться к руководству по MySQL, в котором все они хорошо документированы (находится по адресу http://

[1] Впервые опубликован на сайте http://code.openark.org/blog/mysql/mysql-global-status-difference-using-single-query.

dev.mysql.com/doc/en/ mysqld-option-tables.html). Когда мы обсуждаем набор связанных переменных, имена которых начинаются с общего префикса, то называем эту группу переменными `<prefix>_*`.

Статистика потоков и соединений

В следующих переменных отслеживаются:

- попытки соединения — `Connections`, `Max_used_connections`, `Threads_connected`;
- разорванные соединения — `Aborted_clients`, `Aborted_connects`;
- статистика сетевого трафика — `Bytes_received`, `Bytes_sent`;
- статистика использования потоков — `Slow_launch_threads`, `Threads_cached`, `Threads_created`, `Threads_runnin`.

Если `Aborted_connects` не равна нулю, это свидетельствует, скорее всего, о проблемах в сети или о неудавшейся попытке соединения (например, из-за того, что введены неверный пароль или имя базы данных или, возможно, система мониторинга открыла TCP порт 3306 для проверки того, работает ли сервер). Если это значение становится чрезмерно большим, то возможны серьезные побочные эффекты: MySQL может заблокировать хост.

Имя переменной `Aborted_clients` похоже на имя предыдущей, но смысл ее совершенно иной. Увеличение ее значения обычно связано с ошибками в приложении, в частности, из-за того, что программист забыл корректно закрыть соединение с MySQL перед завершением программы. Как правило, это не является признаком серьезной проблемы.

Состояние записи в двоичный журнал

Переменные состояния `Binlog_cache_use` и `Binlog_cache_disk_use` показывают соответственно, сколько транзакций было сохранено в кэше двоичного журнала и размер скольких транзакций оказался настолько велик, что они не поместились в кэш и их пришлось записывать во временный файл. В MySQL 5.5 есть также переменные `Binlog_stmt_cache_use` и `Binlog_stmt_cache_disk_use`, которые показывают соответствующие показатели для нетранзакционных операторов. Так называемый коэффициент попадания в бинарный журнал обычно не слишком полезен для настройки размера кэша двоичного журнала (см. главу 8 для получения дополнительной информации по этой теме).

Счетчики команд

Переменные `Com_*` подсчитывают, сколько раз вызывались SQL-команды или функции `C API` каждого вида. Например, `Com_select` — это количество команд `SELECT`, а `Com_change_db` показывает, сколько раз изменялась подразумеваемая по умолчанию база данных — то ли с помощью команды `USE`, то ли в результате вызова функции

C API. В переменной Questions[1] подсчитывается общее количество запросов и команд, полученных сервером. Однако оно не равно сумме всех переменных Com_* из-за попадания в кэш запросов, закрытых и разорванных соединений и, возможно, других факторов.

Значение переменной состояния Com_admin_commands может быть очень велико. Она подсчитывает не только административные команды, но и пинг-запросы к самому экземпляру сервера MySQL. Эти запросы выдаются с помощью функции C API и обычно исходят из клиентского кода, например такого (на языке Perl):

```
my $dbh = DBI->connect(...);
while ( $dbh && $dbh->ping ) {
    # Что-то сделать
}
```

Такие ping-запросы считаются мусорными. Быть может, они не слишком нагружают сервер, но все равно впустую транжирят его ресурсы. Нам встречались системы объектно-реляционного отображения (в голову приходит Ruby on Rails), которые пингуют сервер перед каждым запросом, что абсолютно бессмысленно. Пингование сервера, а затем запрос — это классический пример паттерна проектирования «не зная броду, не суйся в воду», который создает гонку. Приходилось нам видеть и библиотеки абстрагирования баз данных, в которых перед каждым запросом подразумеваемая база изменяется, в результате чего значение переменной Com_change_db оказывается очень велико. Того и другого лучше избегать.

Временные таблицы и файлы

Посмотреть переменные, в которых подсчитывается, сколько раз MySQL создавала временные таблицы и файлы, можно такой командой:

```
mysql> SHOW GLOBAL STATUS LIKE 'Created_tmp%';
```

Она показывает статистику по неявным временным таблицам и файлам, которые создаются внутри базы данных для выполнения запросов. В Percona Server также есть команда, которая может отображать явные временные таблицы, которые создают пользователи с помощью команды CREATE TEMPORARY TABLE:

```
mysql> SHOW GLOBAL TEMPORARY TABLES;
```

Операции обработчиков

API обработчиков — это интерфейс между MySQL и ее подсистемами хранения. Переменные Handler_* подсчитывают, например, сколько раз MySQL просила подсистему хранения прочитать следующую строку из индекса. Посмотреть на них можно с помощью такой команды:

```
mysql> SHOW GLOBAL STATUS LIKE 'Handler_%';
```

[1] В MySQL 5.1 эта переменная разделена на две: Questions и Queries с чуть-чуть различающимся смыслом.

Буфер ключей MyISAM

Переменные `Key_*` содержат показатели и счетчики, характеризующие буфер ключей MyISAM. Посмотреть на них можно с помощью команды:

```
mysql> SHOW GLOBAL STATUS LIKE 'Key_%';
```

Дескрипторы файлов

Если вы применяете главным образом подсистему хранения MyISAM, то переменные `Open_*` сообщат, как часто MySQL открывает файлы с расширениями **.frm**, **.MYI** и **.MYD**. InnoDB хранит все данные в файлах табличного пространства, поэтому, если вы используете преимущественно InnoDB, эти переменные не столь важны. Посмотреть на переменные `Open_*` можно с помощью такой команды:

```
mysql> SHOW GLOBAL STATUS LIKE 'Open_%';
```

Кэш запросов

Оценить работу кэша запросов позволяют переменные состояния `Qcache_*`:

```
mysql> SHOW GLOBAL STATUS LIKE 'Qcache_%';
```

Типы команд SELECT

В переменных `Select_*` подсчитывается количество запросов определенных типов `SELECT`. Они полезны, когда нужно оценить соотношение между запросами с разными планами выполнения. К сожалению, для других типов команд, в частности `UPDATE` и `REPLACE`, соответствующих переменных состояния не существует, однако для изучения производительности запросов, отличных от `SELECT`, можно воспользоваться переменными состояния `Handler_*` (см. ранее). Чтобы увидеть все переменные `Select_*`, выполните такую команду:

```
mysql> SHOW GLOBAL STATUS LIKE 'Select_%';
```

На наш взгляд, переменные `Select_*` можно расположить в порядке возрастания затрат на запрос следующим образом.

- `Select_range`. Количество операций соединения, в которых просматривался диапазон индекса для первой таблицы.
- `Select_scan`. Количество операций соединения, в которых выполнялось полное сканирование первой таблицы. В этом нет ничего плохого, если в результат соединения должны войти все строки из первой таблицы, но никуда не годится, если нужны только определенные строки, а для их эффективной выборки нет индекса.
- `Select_full_range_join`. Количество операций соединения, в которых использовалось значение из таблицы n для выборки строк по диапазону индекса для таблицы $n + 1$. В зависимости от запроса затраты могут быть как выше, так и ниже, чем при `Select_scan`.

- `Select_range_check`. Количество операций соединения, в которых для каждой строки из таблицы n приходилось заново оценивать индексы таблицы $n + 1$, чтобы понять, какой из них окажется наименее затратным. Обычно это означает, что у таблицы $n + 1$ нет индексов, полезных для соединения. Издержки для такого плана выполнения очень высоки.
- `Select_full_join`. Количество перекрестных соединений либо соединений, для которых не заданы критерии отбора строк из таблиц. Количество просматриваемых в этом случае строк равно произведению количества строк во всех соединяемых таблицах. Обычно это очень плохо.

Обычно две последние переменные не должны быстро увеличиваться, однако, если так происходит, это может оказаться индикатором плохого запроса. Дополнительную информацию о том, как найти такие запросы, см. в главе 3.

Операции сортировки

Мы много говорили об оптимизации сортировки в нескольких предыдущих главах, поэтому вы уже должны хорошо представлять себе механизм ее работы. Если MySQL не может воспользоваться индексом для выборки строки в нужном порядке, то она вынуждена прибегнуть к файловой сортировке, в результате чего увеличиваются переменные состояния `Sort_*`. Если не считать переменную `Sort_merge_passes`, то повлиять на их значения можно только путем добавления индексов, которые MySQL могла бы задействовать для сортировки. Что касается переменной `Sort_merge_passes`, то она зависит от серверной переменной `sort_buffer_size` (не путайте с серверной переменной `myisam_sort_buffer_size`). MySQL использует буфер сортировки для хранения порции обрабатываемых строк. После сортировки они объединяются с ранее вычисленным результатом, переменная `Sort_merge_passes` увеличивается на 1 и буфер заполняется следующей порцией строк. Однако не стоит использовать эту переменную в качестве руководства при определении размера буфера сортировки, как показано в главе 3.

Посмотреть на переменные `Sort_*` можно с помощью такой команды:

```
mysql> SHOW GLOBAL STATUS LIKE 'Sort_%';
```

MySQL увеличивает переменные `Sort_scan` и `Sort_range`, когда читает отсортированные строки из результата файловой сортировки и возвращает их клиенту. Разница между ними в том, что первая переменная увеличивается, когда план запроса привел к увеличению `Select_scan` (см. предыдущий раздел), а вторая — когда увеличивается `Select_range`. Оба вида сортировки идентичны с точки зрения реализации и затрат — переменные просто отражают различие между типами планов запроса, приведших к сортировке.

Блокирование таблиц

Переменные `Table_locks_immediate` и `Table_locks_waited` говорят о том, сколько блокировок было выдано немедленно, а скольких пришлось ждать. Но имейте в виду, что они отражают статистику блокировки на уровне сервера, а не подсистемы хранения.

Переменные, относящиеся к InnoDB

Переменные `Innodb_*` показывают некоторые данные, выдаваемые командой `SHOW ENGINE INNODB STATUS`, которая рассматривается далее в этом приложении. Эти переменные можно сгруппировать по имени: `Innodb_buffer_pool_*`, `Innodb_log_*` и т. д. Внутренние механизмы InnoDB мы обсудим подробнее, когда будем говорить о команде `SHOW ENGINE INNODB STATUS`.

Эти переменные, появившиеся в версии MySQL 5.0, имеют важный побочный эффект: при их просмотре запрашивается глобальная блокировка, которая удерживается в течение времени обхода всего буферного пула InnoDB. И пока это происходит, остальные потоки блокируются и вынуждены ждать. Это искажает некоторые переменные состояния, например `Threads_running`, значение которой оказывается выше, чем обычно (иногда даже существенно выше, все зависит от того, насколько занят сервер). Тот же эффект проявляется при выполнении команды `SHOW ENGINE INNODB STATUS` и доступе к статистике с помощью таблиц `INFORMATION_SCHEMA` (начиная с версии MySQL 5.0 команды `SHOW STATUS` и `SHOW VARIABLES` реализованы с помощью запросов к таблицам `INFORMATION_SCHEMA`).

Поэтому в упомянутых версиях MySQL такие операции могут оказаться весьма затратными — слишком частые проверки состояния сервера (скажем, раз в секунду) сопряжены с заметными издержками. Использование команды `SHOW STATUS LIKE` не помогает, так как сначала извлекаются все переменные состояния, а уже потом отсеиваются ненужные.

В MySQL 5.5 гораздо больше переменных, чем в 5.1, а еще больше их в Percona Server.

Переменные, относящиеся к подключаемым модулям

Начиная с версии MySQL 5.1, поддерживается новая архитектура подключаемых подсистем хранения и предоставляется механизм для регистрации подсистемами собственных переменных состояния и конфигурации. Такие переменные появятся, если вы подключите соответствующую подсистему хранения. Они всегда начинаются с названия подключаемого модуля.

Команда SHOW INNODB STATUS

Подсистема хранения InnoDB выдает массу информации о своем внутреннем состоянии по команде `SHOW ENGINE INNODB STATUS` или по ее более старому синониму `SHOW INNODB STATUS`.

В отличие от большинства других команд `SHOW`, она выводит одну строку, а не таблицу, состоящую из строк и столбцов. Строка разделена на секции, в каждой из которых представлены сведения о каком-то одном аспекте InnoDB. Часть этой информации представляет интерес в основном для разработчиков InnoDB, но в остальном она весьма полезна и даже необходима любому, кто желает разобраться в работе InnoDB и настроить ее для получения максимальной производительности.

Старые версии InnoDB обычно выводят 64-разрядные числа двумя блоками: старшие 32 бита и младшие 32 бита. Примером может служить идентификатор транзакции: TRANSACTION 0 3793469. Чтобы получить полное 64-разрядное значение, нужно сдвинуть первое число на 32 бита влево и прибавить к нему второе. Мы продемонстрируем эту технику позже.

Среди прочего команда `SHOW ENGINE INNODB STATUS` выводит определенную усредненную статистику, например количество вызовов `fsync()` в секунду. Усреднение производится за период с момента последнего выполнения команды, поэтому, если вы хотите изучить статистику, запрашивайте ее с интервалами примерно 30 секунд, чтобы данные успели накопиться. Не все результаты генерируются строго в один и тот же момент, поэтому средние вычисляются за разные промежутки времени. Кроме того, в InnoDB есть внутренний интервал сброса, он непредсказуем и меняется от версии к версии. Необходимо обращать внимание на то, за какой временной промежуток сгенерированы средние, поскольку в разных выборках он может различаться.

Результат содержит достаточно информации, чтобы при желании вручную вычислить среднее большинство значений. Однако большую помощь может оказать инструмент мониторинга, например программа innotop, который сам вычисляет разности между последовательными выборками и производит усреднение.

Заголовок

Первая секция представляет собой заголовок, который знаменует начало вывода, содержит текущие дату и время, а также показывает, какой интервал прошел с момента последней распечатки. В строке 2 мы видим текущие дату и время. В строке 4 указано, за какой период производилось усреднение: либо со времени последней распечатки, либо с момента внутреннего сброса:

```
1  =====================================
2  070913 10:31:48 INNODB MONITOR OUTPUT
3  =====================================
4  Per second averages calculated from the last 49 seconds
```

Секция SEMAPHORES

Если рабочая нагрузка характеризуется высокой конкурентностью, то имеет смысл обратить внимание на секцию `SEMAPHORES`. В ней приведены данные двух видов: счетчики событий и текущий список ожиданий (его может и не быть). Если вы видите какие-то узкие места, то эта информация поможет определить причину их появления. К сожалению, выявить такие места недостаточно, надо еще понять, что с ними делать, а это уже сложнее. Но некоторые рекомендации мы дадим. Вот пример распечатки:

```
1  ----------
2  SEMAPHORES
3  ----------
4  OS WAIT ARRAY INFO: reservation count 13569, signal count 11421
```

```
5  --Thread 1152170336 has waited at ./../include/buf0buf.ic line 630 for 0.00
     seconds the semaphore:
6  Mutex at 0x2a957858b8 created file buf0buf.c line 517, lock var 0
7  waiters flag 0
8  wait is ending
9  --Thread 1147709792 has waited at ./../include/buf0buf.ic line 630 for 0.00
     seconds the semaphore:
10 Mutex at 0x2a957858b8 created file buf0buf.c line 517, lock var 0
11 waiters flag 0
12 wait is ending
13 Mutex spin waits 5672442, rounds 3899888, OS waits 4719
14 RW-shared spins 5920, OS waits 2918; RW-excl spins 3463, OS waits 3163
```

В строке 4 приводится информация о массиве ожиданий операционной системы — это массив слотов. InnoDB резервирует слоты в массиве семафоров. Семафор — это примитив операционной системы, с помощью которого она сигнализирует потокам, что ожидание закончилось и они могут продолжить работу. В этой строке показано, сколько раз InnoDB пришлось использовать механизм ожиданий операционной системы. Счетчик резервирования (`reservation count`) сообщает, как часто InnoDB выделяла слоты, а счетчик сигналов (`signal count`) — сколько раз потокам посылались сигналы с помощью семафоров из этого массива. Как мы скоро увидим, ожидание с помощью примитивов ОС более затратно, чем активное ожидание (spin wait).

В строках 5–12 показаны потоки InnoDB, которые в данный момент ожидают освобождения мьютекса. В этом примере мы видим два ожидания, каждое из которых начинается словами `-- Thread <num> has waited...`. Если бы рабочая нагрузка не характеризовалась высокой конкурентностью, заставляющей InnoDB прибегать к услугам ОС, то эта секция была бы пуста. Если вы знакомы с исходным кодом InnoDB, весьма полезной информацией станет имя файла, в котором поток ожидает события. Оно позволяет понять, в каком месте InnoDB возникают горячие точки. Например, если много потоков ожидают в файле `buf0buf.ic`, то имеется конкуренция за буферный пул. В распечатке также показано, как долго поток ждет, а поле `waiters flag` говорит о том, сколько потоков стоят в очереди к мьютексу.

Фраза `wait is ending` означает, что мьютекс уже освободился, но операционная система еще не запланировала выполнение потока.

Может возникнуть вопрос: чего именно ждет InnoDB? Эта подсистема применяет мьютексы и семафоры для защиты критических участков кода, разрешая их выполнение лишь одному потоку одновременно или блокируя выполняющие операции записи потоков при наличии потоков читающих и т. д. В коде InnoDB имеется множество критических секций, и при определенных условиях в распечатке может появиться любая из них. Чаще всего речь идет о получении доступа к странице буферного пула.

После списка ожидающих потоков, в строках 13 и 14, мы видим дополнительные счетчики событий. В строке 13 показаны несколько счетчиков, относящихся к мьютексам, а в строке 14 — к разделяемым и монопольным блокировкам чтения/записи. И в том и в другом случае демонстрируется, сколько раз InnoDB прибегала к ожиданию на уровне ОС.

В InnoDB применяется многоступенчатая стратегия ожидания. Сначала она пытается дождаться блокировки с помощью активного ожидания. Если после заранее заданного числа итераций (определенных конфигурационным параметром `innodb_sync_spin_loops`) получить блокировку не удалось, она обращается к более дорогому и сложному массиву ожиданий[1].

Активное ожидание, как правило, не очень затратно, но на него расходуется время процессора, поскольку программа постоянно проверяет, можно ли заблокировать ресурс. Это не так плохо, как может показаться, поскольку обычно у процессора есть свободные циклы, пока он ожидает завершения ввода/вывода. И даже если свободных циклов нет, все равно активное ожидание гораздо менее затратно, чем альтернативные способы. Однако опрос в цикле монополизирует процессор, не допуская к нему другие потоки, в которых, возможно, нашлась бы полезная работа.

Альтернативой активному ожиданию является контекстное переключение на уровне операционной системы, в результате которого другой поток получает возможность поработать, пока первый ждет. Спящий поток будет разбужен сигналом от семафора в массиве ожиданий. Сигнализация с помощью семафора реализована эффективно, но вот контекстное переключение — очень затратная операция. Причем она производит кумулятивный эффект: тысячи контекстных переключений могут привести к ощутимым издержкам.

Можно попытаться соблюсти баланс между активными ожиданиями и ожиданиями на уровне ОС, изменив системную переменную `innodb_sync_spin_loops`. Если количество активных ожиданий в секунду не слишком велико (порядка сотен или тысяч), то можно о них особо не беспокоиться. Это решение обычно нужно принимать, анализируя исходный код или консультируясь с экспертами. Вы также можете рассмотреть использование Performance Schema или посмотреть на результаты работы команды `SHOW ENGINE INNODB MUTEX`.

Секция LATEST FOREIGN KEY ERROR

Следующая секция, `LATEST FOREIGN KEY ERROR`, появляется только в том случае, если появились ошибки внешнего ключа. В исходном коде много мест, где могут генерироваться такие ошибки, причем причины у них бывают разные. Иногда это родительские либо дочерние строки, которые транзакция искала в попытке вставить, обновить или удалить запись. А иногда виновато рассогласование между таблицами, обнаруженное при попытке добавить или удалить внешний ключ. Также проблема может возникнуть из-за изменения структуры таблицы, в которой уже есть внешний ключ.

Информация, приведенная в этой секции, очень полезна для выяснения точных причин сообщений об ошибках внешнего ключа, которые зачастую звучат туманно.

[1] В версии MySQL 5.1 код массива ожиданий был переписан и стал гораздо более эффективным.

Рассмотрим несколько примеров. Для начала создадим две таблицы, связанные внешним ключом, и вставим в них данные:

```
CREATE TABLE parent (
    parent_id int NOT NULL,
    PRIMARY KEY(parent_id)
 ) ENGINE=InnoDB;
CREATE TABLE child (
    parent_id int NOT NULL,
    KEY parent_id (parent_id),
    CONSTRAINT child_ibfk_1 FOREIGN KEY (parent_id) REFERENCES parent (parent_id)
 ) ENGINE=InnoDB;

INSERT INTO parent(parent_id) VALUES(1);
INSERT INTO child(parent_id) VALUES(1);
```

Существуют два широких класса ошибок внешнего ключа. В первый попадают ошибки, которые возникают при добавлении, обновлении и удалении данных таким образом, что нарушается ограничение внешнего ключа. Например, вот что произойдет, если мы удалим строку из родительской таблицы:

```
DELETE FROM parent;
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint
fails (`test/child`, CONSTRAINT `child_ibfk_1` FOREIGN KEY (`parent_id`) REFERENCES
`parent` (`parent_id`))
```

Текст здесь понятен, такие сообщения выдаются при любой попытке добавить, обновить или удалить строки так, что это приведет к рассогласованию. Вот что покажет в этом случае команда `SHOW ENGINE INNODB STATUS`:

```
1  ------------------------
2  LATEST FOREIGN KEY ERROR
3  ------------------------
4  070913 10:57:34 Transaction:
5  TRANSACTION 0 3793469, ACTIVE 0 sec, process no 5488, OS thread id 1141152064
   updating or deleting, thread declared inside InnoDB 499
6  mysql tables in use 1, locked 1
7  4 lock struct(s), heap size 1216, undo log entries 1
8  MySQL thread id 9, query id 305 localhost baron updating
9  DELETE FROM parent
10 Foreign key constraint fails for table `test/child`:
11 '
12   CONSTRAINT `child_ibfk_1` FOREIGN KEY (`parent_id`) REFERENCES `parent`
   (`parent_     id`)
13 Trying to delete or update in parent table, in index `PRIMARY` tuple:
14 DATA TUPLE: 3 fields;
15  0: len 4; hex 80000001; asc     ;; 1: len 6; hex 00000039e23d; asc    9 =;;
   2: len     7; hex 000000002d0e24;  asc     - $;;
16
17 But in child table `test/child`, in index `parent_id`, there is a record:
18 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
19  0: len 4; hex 80000001; asc     ;; 1: len 6; hex 000000000500; asc      ;;
```

В строке 4 демонстрируются дата и время последней ошибки, связанной с внешним ключом. В строках 5–9 приведены подробные сведения о транзакции, в которой произошла ошибка (мы поясним их смысл чуть позже). В строках 10–19 точно

описывается, что именно пыталась изменить InnoDB, когда обнаружила проблему. В основном это данные строки, приведенные к печатаемому виду (об этом мы тоже еще поговорим).

Есть еще один класс ошибок внешнего ключа, которые отлаживать куда труднее. Вот что произойдет, если мы попытаемся изменить структуру родительской таблицы:

```
ALTER TABLE parent MODIFY parent_id INT UNSIGNED NOT NULL;
ERROR 1025 (HY000): Error on rename of './test/#sql-1570_9' to './test/parent'
(errno: 150)
```

Это сообщение уже не так понятно, но `SHOW ENGINE INNODB STATUS` проливает свет на причину ошибки:

```
1  ------------------------
2  LATEST FOREIGN KEY ERROR
3  ------------------------
4  070913 11:06:03 Error in foreign key constraint of table test/child:
5  there is no index in referenced table which would contain
6  the columns as the first columns, or the data types in the
7  referenced table do not match to the ones in table. Constraint:
8  ,
9    CONSTRAINT child_ibfk_1 FOREIGN KEY (parent_id) REFERENCES parent (parent_id)
10 The index in the foreign key in table is parent_id
11 See http://dev.mysql.com/doc/refman/5.0/en/innodb-foreign-key-constraints.html
12 for correct foreign key definition.
```

В данном случае она возникла из-за различных типов данных. Столбцы, связанные внешним ключом, должны иметь одинаковые типы данных, включая все модификаторы (в частности, `UNSIGNED`, из-за которого и произошла ошибка). Если вы видите ошибку 1025 и не понимаете, в чем дело, то начинать расследование нужно с изучения распечатки `SHOW ENGINE INNODB STATUS`.

Сообщения об ошибках внешнего ключа перезаписываются каждый раз, когда появляется новая ошибка. Инструмент pt-fk-error-logger из пакета Percona Toolkit поможет вам сохранить их для последующего анализа.

Секция LATEST DETECTED DEADLOCK

Как и секция, относящаяся к внешним ключам, секция `LATEST DETECTED DEADLOCK` присутствует только в случае, если сервер обнаружил взаимоблокировку. Сообщения об ошибках взаимоблокировки также перезаписываются каждый раз, когда появляется новая ошибка, а инструмент pt-deadlock-logger от Percona Toolkit поможет вам сохранить их для последующего анализа.

Взаимоблокировка возникает при наличии цикла в графе ожиданий (wait-for graph), где представлены строки, на которые удерживается блокировка, и строки, для которых ожидается освобождение блокировки. Этот цикл может быть произвольно большим. InnoDB обнаруживает взаимоблокировки мгновенно, так как проверяет наличие цикла всякий раз, когда транзакция вынуждена ждать освобождения блокировки строки. Взаимоблокировки могут быть весьма сложными, но в этой секции показаны только последние две участвующие транзакции, завершающая команда

в каждой из них и блокировки, образовавшие цикл в графе. Вы не увидите ни других транзакций, которые, возможно, вносят свой вклад в цикл, ни команды, которая в действительности захватила блокировки, если она была не последней. Тем не менее изучение этой распечатки обычно помогает выявить причину взаимоблокировки.

На самом деле в InnoDB есть две разновидности взаимоблокировок. Первая, знакомая многим, связана с наличием реального цикла в графе ожиданий. Вторая возникает, когда граф ожиданий настолько велик, что искать в нем циклы слишком затратно. Если требуется проверить граф, содержащий более 1 миллиона блокировок, или во время проверки транзакций глубина рекурсии превысит 200, то InnoDB сдается и считает, что взаимоблокировка имеется. Эти константы зашиты в код InnoDB и не конфигурируются (хотя при желании их можно изменить, перекомпилировав исходный код InnoDB). О наличии такой взаимоблокировки свидетельствует сообщение в распечатке `TOO DEEP OR LONG SEARCH IN THE LOCK TABLE WAITS-FOR GRAPH` («Слишком глубокий или продолжительный поиск в графе ожиданий блокировок»).

InnoDB выводит не только номера транзакций и блокировки, которые они удерживают или ожидают, но и сами записи. Эта информация полезна главным образом разработчикам InnoDB, но в настоящее время отключить ее вывод невозможно. К сожалению, она может заполнить все место, отведенное для распечатки, так что вы не увидите последующие секции. Единственный способ как-то поправить ситуацию — сделать так, чтобы вместо большой взаимоблокировки возникала маленькая, или использовать Percona Server, в котором добавлены переменные конфигурации, запрещающие вывод слишком подробного текста.

Далее приведен пример сведений о взаимоблокировке:

```
1 ------------------------
2 LATEST DETECTED DEADLOCK
3 ------------------------
4 070913 11:14:21
5 *** (1) TRANSACTION:
6 TRANSACTION 0 3793488, ACTIVE 2 sec, process no 5488, OS thread id 1141287232
  starting index read
7 mysql tables in use 1, locked 1
8 LOCK WAIT 4 lock struct(s), heap size 1216
9 MySQL thread id 11, query id 350 localhost baron Updating
10 UPDATE test.tiny_dl SET a = 0 WHERE a <> 0
11 *** (1) WAITING FOR THIS LOCK TO BE GRANTED:
12 RECORD LOCKS space id 0 page no 3662 n bits 72 index `GEN_CLUST_INDEX` of table
   `test/tiny_dl` trx id 0 3793488 lock_mode X waiting
13 Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
14 0: len 6; hex 000000000501 ...[ опущено ] ...
15
16 *** (2) TRANSACTION:
17 TRANSACTION 0 3793489, ACTIVE 2 sec, process no 5488, OS thread id 1141422400
   starting index read, thread declared inside InnoDB 500
18 mysql tables in use 1, locked 1
19 4 lock struct(s), heap size 1216
20 MySQL thread id 12, query id 351 localhost baron Updating
21 UPDATE test.tiny_dl SET a = 1 WHERE a <> 1
22 *** (2) HOLDS THE LOCK(S):
23 RECORD LOCKS space id 0 page no 3662 n bits 72 index `GEN_CLUST_INDEX` of table
   `test/tiny_dl` trx id 0 3793489 lock mode S
```

```
24 Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
25 0: ... [ опущено ] ...
26
27 *** (2) WAITING FOR THIS LOCK TO BE GRANTED:
28 RECORD LOCKS space id 0 page no 3662 n bits 72 index `GEN_CLUST_INDEX` of table
   `test/tiny_dl` trx id 0 3793489 lock_mode X waiting
29 Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
30 0: len 6; hex 000000000501 ...[ опущено ] ...
31
32 *** WE ROLL BACK TRANSACTION (2)
```

В строке 4 показано, когда произошла взаимоблокировка, а в строках 5–10 приведены сведения о первой участвующей в ней транзакции. Назначение отдельных полей мы объясним в следующем разделе.

В строках 11–15 показаны блокировки, которых ожидала транзакция 1 в момент, когда была обнаружена взаимоблокировка. Кое-какую информацию в строке 14, полезную только для отладки InnoDB, мы опустили. Обратите особое внимание на строку 12, в которой говорится, что транзакция запросила монопольную (`X`) блокировку на индекс `GEN_CLUST_INDEX`[1] по таблице `test.tiny_dl`.

В строках 16–21 показано состояние второй транзакции, а в строках 22–26 — список удерживаемых ею блокировок. В строке 25 перечислялись несколько записей, которые мы для краткости опустили. Одной из них как раз была запись, чьего освобождения ожидала первая транзакция. Наконец, в строках 27–31 показаны блокировки, которых ожидала вторая транзакция.

Цикл в графе ожиданий образовался из-за того, что каждая транзакция удерживает блокировку, нужную другой транзакции. InnoDB не показывает все удерживаемые и ожидаемые блокировки, но выводит достаточно информации, чтобы можно было понять, какие индексы использовались при выполнении запросов. Это помогает решить, можно ли что-то сделать для предотвращения взаимоблокировок.

Если удастся добиться того, чтобы оба запроса просматривали один и тот же индекс в одном и том же направлении, то количество взаимоблокировок сократится, так как при запросе блокировок в одном и том же порядке цикл никогда не возникает. Иногда сделать это несложно. Например, если в транзакции нужно обновить несколько записей, то отсортируйте их по первичному ключу в памяти приложения и обновляйте именно в таком порядке — взаимоблокировки не будет. Но в других случаях ничего не получится — например, когда два процесса работают с одной таблицей, используя при этом разные индексы.

В строке 32 показано, какая транзакция выбрана жертвой взаимоблокировки. InnoDB выбирает ту транзакцию, которую с ее точки зрения будет проще откатить, то есть в которой количество обновлений наименьшее.

Очень полезно просмотреть общий журнал, найти в нем все запросы, выполненные в тех потоках, которые участвовали в инциденте, и выяснить истинную причину взаимоблокировки. В следующем разделе рассказано о том, где искать идентификатор потока в распечатке сведений о взаимоблокировке.

---

[1] Этот индекс InnoDB создает самостоятельно, если не задан первичный ключ.

Секция TRANSACTIONS

В этой секции содержится сводная информация о транзакциях InnoDB, за которой следует список активных транзакций. Вот первые несколько строчек (заголовок):

```
1 ------------
2 TRANSACTIONS
3 ------------
4 Trx id counter 0 80157601
5 Purge done for trx's n:o <0 80154573 undo n:o <0 0
6 History list length 6
7 Total number of lock structs in row lock hash table 0
```

Состав распечатки зависит от версии MySQL, но в любом случае имеется по крайней мере следующая информация.

- Строка 4 — текущий идентификатор транзакции. Это системная переменная, увеличивающаяся на единицу для каждой новой транзакции.
- Строка 5 — идентификатор транзакции, для которой InnoDB удалила старые MVCC-версии строк. Узнать, сколько старых версий еще не удалено, можно, посмотрев на разность между этим значением и текущим идентификатором транзакции. Не существует четкого и однозначного правила, насколько большое значение этой величины можно считать безопасным. Если никто не обновляет никаких данных, то большое значение еще не означает, что скопилось много неудаленных версий, потому что все транзакции на самом деле видят одну и ту же версию базы. Но если обновляется много строк, то одна или даже несколько версий каждой строки остаются в памяти. Наилучшая стратегия сокращения издержек — делать коммит транзакции сразу после завершения работы, а не оставлять их открытыми на длительное время, поскольку даже если незакоммиченая транзакция ничего не делает, она все равно не дает InnoDB удалить старые версии строк.

  Также в строке 5 указан номер записи в журнале отмены, с которым в настоящий момент работает процесс очистки InnoDB. Если он равен `0 0`, как в данном примере, значит, процесс очистки простаивает.
- Строка 6 — длина списка истории, то есть количество еще не удаленных транзакций в пространстве отмены в файлах данных InnoDB. Когда транзакция производит обновление и коммитится, это число увеличивается. Как только процесс очистки удаляет старые версии, оно уменьшается. Процедура очистки также обновляет значение, показанное в строке 5.
- Строка 7 — количество lock-структур. В каждой lock-структуре обычно хранится несколько блокировок уровня строки, так что это не то же самое, что количество заблокированных строк.

После заголовка идет список транзакций. В текущих версиях MySQL вложенные транзакции не поддерживаются, поэтому в каждый момент времени в рамках одного соединения может существовать не более одной транзакции и каждая транзакция принадлежит ровно одному соединению. Под каждую транзакцию в этой распечатке отведено по меньшей мере две строчки. Вот пример минимальной информации о транзакции:

```
1 ---TRANSACTION 0 3793494, not started, process no 5488, OS thread id 1141152064
2 MySQL thread id 15, query id 479 localhost baron
```

Первая строчка начинается с идентификатора и состояния транзакции. Эта транзакция не начата, то есть закоммичена и не инициировала никаких команд, которые могли бы повлиять на другие транзакции, — скорее всего, она просто ничего не делает. Далее следует информация о процессе и потоке. Во второй строчке показан идентификатор процесса MySQL — то же самое число, которое демонстрируется в столбце `Id` в списке процессов, выдаваемых командой `SHOW FULL PROCESSLIST`. Затем идут внутренний номер запроса и сведения о подключении, также печатаемые командой `SHOW FULL PROCESSLIST`.

Однако о транзакции может быть приведено гораздо больше информации. Вот более сложный пример:

```
1 ---TRANSACTION 0 80157600, ACTIVE 4 sec, process no 3396, OS thread id 1148250464,
  thread declared inside InnoDB 442
2 mysql tables in use 1, locked 0
3 MySQL thread id 8079, query id 728899 localhost baron Sending data
4 select sql_calc_found_rows * from b limit 5
5 Trx read view will not see trx with id>= 0 80157601, sees <0 80157597
```

Из строки 1 видно, что транзакция была активна 4 секунды. Возможны следующие состояния: not started («не начата»), active («активна»), prepared («подготовлена») и committed in memory («закоммичена в памяти»). После того как транзакция будет закоммичена на диске, она переходит в состояние not started. Дополнительно может быть выведена информация о том, чем сейчас занята транзакция, хотя в данном примере этого нет. В исходном коде встречается свыше 30 строковых констант, которые могут быть указаны в этом месте, например: fetching rows («выборка строк»), adding foreign keys («добавление внешних ключей») и т. д.

Фраза `thread declared inside InnoDB 442` («поток находится внутри InnoDB 442») в строке 1 означает, что поток занят какой-то операцией внутри ядра InnoDB и у него осталось еще 442 неиспользованных «билета». Другими словами, этому SQL-запросу разрешено войти в ядро InnoDB еще 442 раза. Система «билетов» ограничивает количество одновременно выполняющихся внутри ядра потоков с целью предотвратить их пробуксовку на некоторых платформах. Даже если показано состояние потока inside InnoDB, это еще не означает, что он выполняет всю работу внутри подсистемы хранения, — некоторые операции могут выполняться на уровне сервера и лишь временами тем или иным образом взаимодействовать с InnoDB. Иногда можно увидеть, что транзакция находится в состоянии sleeping before joining InnoDB queue («спит перед постановкой в очередь InnoDB») или waiting in InnoDB queue («ждет в очереди InnoDB»).

В следующей строчке может содержаться информация о том, сколько таблиц используется и заблокировано текущей командой. Обычно InnoDB не блокирует таблицы, но для некоторых команд это делается. Заблокированные таблицы могут появляться и тогда, когда сервер MySQL выполнил блокировку на более высоком уровне, чем InnoDB. Если транзакция заблокировала какие-то строки, то будет присутствовать информация о количестве lock-структур (еще раз подчеркнем, что это не то же самое, что количество блокировок строк) и размере кучи (пример был показан ранее в распечатке сведений о взаимоблокировке).

Размер кучи — это объем памяти, отведенной для хранения блокировок строк. В InnoDB блокировки строк реализованы с помощью специальной таблицы битовых векторов, которая теоретически позволяет представлять блокировку всего одним битом. Наши тесты показывают, что в общем случае на блокировку отводится не более 4 бит.

Строка 3 в этом примере содержит чуть больше информации, чем строка 2 — в предыдущем: в конце мы видим состояние потока Sending data («отправка данных»). Это именно то, что показывает команда `SHOW FULL PROCESSLIST` в столбце `Command`.

Если транзакция в данный момент занята выполнением запроса, то далее будет приведен текст этого запроса (в некоторых версиях MySQL только его часть), как в данном случае в строке 4.

В строке 5 показано представление базы (read view), используемое транзакцией, то есть диапазоны идентификаторов транзакций, которые наверняка видны и наверняка не видны данной транзакции благодаря механизму многоверсионности. В данном случае между двумя диапазонами имеется промежуток в четыре транзакции, относительно которых нельзя точно сказать, видны они или нет. При выполнении запроса InnoDB должна проверить, видны ли строки, идентификаторы транзакций для которых попадают в этот промежуток.

Если транзакция ожидает блокировки, то после запроса будет приведена информация об этой блокировке. Такие примеры встречаются в разделе о взаимоблокировках ранее. К сожалению, распечатка ничего не говорит о том, какие транзакции удерживают те блокировки, которых ожидает данная транзакция. Вы можете обнаружить их в таблицах `INFORMATION_SCHEMA` в MySQL 5.1 и более новых версиях, если используете плагин InnoDB.

Если транзакций много, то InnoDB может ограничить их количество в распечатке, чтобы объем вывода оказался не слишком велик. В таком случае вы увидите строку `...truncated...` («...обрезано...»).

Секция FILE I/O

В секции `FILE I/O` представлены сведения о состоянии вспомогательных потоков ввода/вывода, а также счетчики производительности:

```
1  --------
2  FILE I/O
3  --------
4  I/O thread 0 state: waiting for i/o request (insert buffer thread)
5  I/O thread 1 state: waiting for i/o request (log thread)
6  I/O thread 2 state: waiting for i/o request (read thread)
7  I/O thread 3 state: waiting for i/o request (write thread)
8  Pending normal aio reads: 0, aio writes: 0,
9   ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0
10  Pending flushes (fsync) log: 0; buffer pool: 0
11  17909940 OS file reads, 22088963 OS file writes, 1743764 OS fsyncs 12  0.20
reads/s, 16384 avg bytes/read, 5.00 writes/s, 0.80 fsyncs/s
```

В строках 4–7 приведена информация о состоянии вспомогательных потоков ввода/вывода. В строках 8–10 показано количество еще не завершенных операций для каждого такого потока, а также количество ожидающих процедур `fsync()` для потоков журнала и пула буферов. Аббревиатура aio означает «асинхронный ввод/вывод» (asynchronous I/O). В строке 11 демонстрируется количество выполненных операций чтения, записи и `fsync()`. Абсолютные значения зависят от рабочей нагрузки, поэтому важнее следить за тем, как они изменяются во времени. В строке 12 указано количество операций в секунду, усредненное по интервалу, который указан в секции заголовка.

Значения `pending` («незавершенные») в строках 8 и 9 — удобный способ выявить приложения, занимающиеся преимущественно вводом/выводом. В этом случае для большинства видов ввода/вывода будут присутствовать незавершенные операции.

На платформе Windows количество вспомогательных потоков ввода/вывода можно регулировать с помощью конфигурационной переменной `innodb_file_io_threads`, поэтому одновременно могут присутствовать несколько потоков чтения и записи. Однако на любой платформе будут существовать по крайней мере следующие четыре потока:

- Insert buffer thread (поток буфера вставки). Отвечает за объединение с буфером вставки (то есть за перенос записей из буфера вставки в табличное пространство);
- Log thread (поток журнала). Отвечает за асинхронный сброс журнала на диск;
- Read thread (поток чтения). Выполняет операции упреждающего чтения для выборки тех данных, которые, как считает InnoDB, понадобятся в ближайшем будущем;
- Write thread (поток записи). Сбрасывает на диск «грязные» буферы.

Секция INSERT BUFFER AND ADAPTIVE HASH INDEX

В этой секции отображается состояние буфера вставки и адаптивного хеш-индекса:

```
1  -------------------------------------
2  INSERT BUFFER AND ADAPTIVE HASH INDEX
3  -------------------------------------
4  Ibuf for space 0: size 1, free list len 887, seg size 889, is not empty
5  Ibuf for space 0: size 1, free list len 887, seg size 889,
6  2431891 inserts, 2672643 merged recs, 1059730 merges
7  Hash table size 8850487, used cells 2381348, node heap has 4091 buffer(s)
8  2208.17 hash searches/s, 175.05 non-hash searches/s
```

В строке 4 приведены информация о размере буфера вставки, длина его списка свободных блоков и размер сегмента. Фраза `for space 0` вроде бы указывает на наличие нескольких буферов вставки — по одному на каждое табличное пространство, но это так и не было реализовано, поэтому в последних версиях MySQL фраза удалена. Поскольку существует всего один буфер вставки, строка 5 — лишняя. В строке 6 показана статистика работы с буфером. Отношение количества объединений (merges) к количеству вставок (inserts) дает представление об эффективности буфера.

В строке 7 демонстрируется состояние адаптивного хеш-индекса. Из строки 8 мы узнаем, сколько операций с этим индексом было произведено за время, указанное

в секции заголовка. Отношение количества поисков по хеш-индексу к количеству поисков без использования хеш-индекса приведено для справки, поскольку средств для конфигурирования адаптивного хеш-индекса нет.

Секция LOG

В этой секции собрана статистика о подсистеме работы с журналом транзакций (журналом обновлений) в InnoDB:

```
1  ---
2  LOG
3  ---
4  Log sequence number 84 3000620880
5  Log flushed up to   84 3000611265
6  Last checkpoint at  84 2939889199
7  0 pending log writes, 0 pending chkp writes
8  14073669 log i/o's done, 10.90 log i/o's/second
```

В строке 4 показан текущий порядковый номер журнала, а в строке 5 — место, до которого журнал был сброшен на диск. Порядковый номер — это просто количество байтов, записанных в файлы журнала, зная его, можно вычислить, сколько данных из буфера журнала еще не сброшено в файлы. В данном случае эта величина равна 9615 байт (13 000 620 880 – 13 000 611 265). В строке 6 показана последняя контрольная точка (она определяет момент времени, когда данные и файлы журналов находились в известном состоянии, пригодном для восстановления). Если последняя контрольная точка слишком сильно отстает от порядкового номера журнала, а разница сопоставима с размером файлов журнала, InnoDB начнет яростный сброс, что очень плохо для производительности. В строках 7 и 8 отображаются количество незавершенных операций с журналом и статистика — ее можно сравнить с величинами в секции `FILE I/O` и узнать, какая доля ввода/вывода приходится на подсистему работы с журналом, а не обусловлена какими-то другими причинами.

Секция BUFFER POOL AND MEMORY

В этой секции приведена статистика использования пула буферов и памяти InnoDB (о настройке пула буферов см. главу 6).

```
1  ----------------------
2  BUFFER POOL AND MEMORY
3  ----------------------
4  Total memory allocated 4648979546; in additional pool allocated 16773888
5  Buffer pool size   262144
6  Free buffers       0
7  Database pages       258053
8  Modified db pages  37491
9  Pending reads 0
10  Pending writes: LRU 0, flush list 0, single page 0
11  Pages read 57973114, created 251137, written 10761167
12  9.79 reads/s, 0.31 creates/s, 6.00 writes/s
13  Buffer pool hit rate 999 / 1000
```

В строке 4 демонстрируется общий объем памяти, выделенной InnoDB, и указано, какая часть этой памяти отведена под дополнительный пул. Дополнительный пул памяти — это просто объем памяти, как правило, небольшой, который выделяет InnoDB, когда хочет использовать собственный распределитель внутренней памяти. Современные версии InnoDB обычно используют распределитель памяти операционной системы, но более старые версии имеют собственный распределитель, поскольку не все операционные системы обеспечивают приемлемую реализацию.

В строках 5–8 приведены различные показатели буферного пула, измеряемые в страницах: размер пула, количество свободных страниц, количество страниц, отведенных для хранения страниц базы данных, и количество «грязных» страниц базы. Часть страниц в пуле буферов подсистема хранения использует для индексов блокировок, адаптивного хеш-индекса и других системных структур, поэтому количество страниц базы данных в пуле никогда не совпадает с общим его размером.

В строках 9 и 10 показано число незавершенных операций чтения и записи (то есть операций, которые InnoDB еще предстоит выполнить в буферном пуле). Эти значения не совпадают со значениями в секции `FILE I/O`, так как InnoDB может объединять несколько логических процедур в одну физическую. Аббревиатура LRU означает least recently used («наиболее давно использованный») — это алгоритм освобождения места для часто используемых страниц пула буферов за счет вытеснения и сброса на диск тех, что задействуются редко. Список сброса (flush list) содержит старые страницы, которые процесс, создающий контрольные точки, должен сбросить на диск при установлении следующей контрольной точки, а показатель «запись одиночных страниц» (single page) относится к независимым операциям записи, которые не удалось объединить.

В строке 8 мы видим, что буферный пул включает 37 491 «грязную» страницу, которые в какой-то момент нужно будет сбросить на диск (то есть они уже модифицированы в памяти, но еще не сохранены). Однако строка 10 говорит, что в данный момент операция сброса не запланирована. Ничего страшного, InnoDB сбросит их, когда ей это будет нужно. Если вы видите большое количество ожидающих операций ввода/вывода где-либо в выводе состояния InnoDB, то, как правило, это симптом довольно серьезной проблемы.

В строке 11 показано, сколько страниц InnoDB прочитала, создала и записала. Количество прочитанных и записанных страниц относится к данным, которые были прочитаны соответственно с диска в буферный пул и обратно. Созданные страницы — это те, для которых InnoDB выделила место в буфере, не читая их содержимое из файла данных, потому что это содержимое ей безразлично (например, они могли принадлежать таблице, которая уже удалена).

В строке 13 показан коэффициент попаданий в кэш, он говорит о том, как часто InnoDB находит нужные страницы в буферном пуле. Он учитывает попадания с момента последней распечатки состояния InnoDB, поэтому если с тех пор сервер ничего не делал, то вы увидите сообщение `No buffer pool page gets since the last printout` («С момента последней распечатки не было обращений к страницам из пула буферов»). Эта информация не слишком полезна с точки зрения определения размера буферного пула.

В MySQL 5.5 может быть несколько буферных пулов, и в распечатке каждому из них будет отведен свой раздел. Percona XtraDB также будет печатать более подробный вывод, например показывая, где именно выделяется память.

Секция ROW OPERATIONS

В этой секции собрана информация об операциях со строками и прочая статистика InnoDB:

```
1  --------------
2  ROW OPERATIONS
3  --------------
4  0 queries inside InnoDB, 0 queries in queue
5  1 read views open inside InnoDB
6  Main thread process no. 10099, id 88021936, state: waiting for server activity
7  Number of rows inserted 143, updated 3000041, deleted 0, read 24865563
8  0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s
9  ----------------------------
10  END OF INNODB MONITOR OUTPUT
11  ============================
```

В строке 4 показано, сколько потоков сейчас находится в ядре InnoDB (мы уже касались этой темы при обсуждении секции `TRANSACTIONS`). Запросы в очереди (queries in queue) — это те потоки InnoDB, которые пока не допущены в ядро, чтобы ограничить общее количество одновременно работающих потоков. Ранее мы также отмечали, что перед постановкой в очередь запрос может некоторое время спать.

В строке 5 указывается количество открытых в настоящий момент представлений базы (read view). Представление базы — это согласованный снимок многоверсионного содержимого базы данных на момент начала транзакции. Посмотреть, имеет ли конкретная транзакция представление базы, можно в секции `TRANSACTIONS`.

В строке 6 отображается состояние главного потока ядра. Оно может принимать следующие значения:

- `doing background drop tables` — фоновое удаление таблиц;
- `doing insert buffer merge` — объединение с буфером вставки;
- `flushing buffer pool pages` — сброс страниц из буферного пула;
- `flushing log` — сброс журнала;
- `making checkpoint` — запись контрольной точки;
- `purging` — очистка;
- `reserving kernel mutex` — резервирование мьютекса ядра;
- `sleeping` — сон;
- `suspending` — приостановка;
- `waiting for buffer pool flush to end` — ожидание завершения сброса буферного пула;
- `waiting for server activity` — ожидание действия со стороны сервера.

На большинстве серверов вы, как правило, должны видеть значение «спящий». Если вы сделали несколько снимков и неоднократно видели другое состояние, например «сброс страниц из буферного пула», можно заподозрить наличие какой-то связанной с этой деятельностью проблемы, например проблемы яростного сброса, причиной которой может быть плохой алгоритм сброса в конкретной версии InnoDB или плохая конфигурация, допустим слишком маленькие файлы журналов транзакций.

В строках 7 и 8 приведены статистика количества вставленных, обновленных, удаленных и прочитанных строк, а также усредненные значения количества этих операций в секунду. За ними имеет смысл наблюдать, если вы хотите знать, чем занимается InnoDB.

Распечатка, выдаваемая командой `SHOW ENGINE INNODB STATUS`, заканчивается строками 9–11. Если вы их не видите, то, скорее всего, произошла большая взаимоблокировка, из-за которой пришлось обрезать распечатку.

Команда SHOW PROCESSLIST

Список процессов — это список соединений, или потоков, установленных с MySQL в данный момент времени. Команда `SHOW PROCESSLIST` выводит перечень всех потоков, включая в него сведения об их состоянии, например:

```
mysql> SHOW FULL PROCESSLIST\G
*************************** 1. row ***************************
      Id: 61539
    User: sphinx
    Host: se02:58392
      db: art136
 Command: Query
    Time: 0
   State: Sending data
    Info: SELECT a.id id, a.site_id site_id, unix_timestamp(inserted) AS
inserted,forum_id, unix_timestamp(p
*************************** 2. row ***************************
      Id: 65094
    User: mailboxer
    Host: db01:59659
      db: link84
 Command: Killed
    Time: 12931
   State: end
 Info: update link84.link_in84 set url_to = replace(replace(url_to,'&amp;','&'),
'%20','+'), url_prefix=repl
```

Существует несколько инструментов, в том числе innotop, умеющих показывать список процессов в динамике.

Вы также можете получить эту информацию из таблицы в `INFORMATION_SCHEMA`. Percona Server и MariaDB добавляют в эту таблицу более полезную информацию, такую как столбец времени с высокоточными значениями или столбец, указывающий, сколько работы выполнил запрос, который можно использовать в качестве индикатора прогресса.

В столбцах `Command` и `State` отображается состояние потока. Обратите внимание на то, что первый процесс выполняет запрос и отправляет данные, тогда как второй был прерван (`Killed`), возможно, потому, что работал слишком долго и кто-то сознательно остановил его командой `KILL`. Поток может оставаться в этом состоянии в течение определенного периода, потому что прерывание не происходит мгновенно. Например, для отката начатой в потоке транзакции требуется время.

Команда `SHOW FULL PROCESSLIST` (с дополнительным ключевым словом `FULL`) показывает полный текст запроса, который в противном случае усекается до 100 символов.

Команда SHOW ENGINE MUTEX STATUS

Эта команда возвращает подробную информацию о мьютексах InnoDB и полезна главным образом для разрешения проблем с масштабируемостью и конкурентностью. Как уже объяснялось, каждый мьютекс защищает одну критическую секцию программы.

Состав распечатки зависит от версии MySQL и параметров, заданных при компиляции. Далее приведен пример для сервера MySQL 5.5:

```
mysql> SHOW ENGINE INNODB MUTEX;
+--------+-----------------------------+-------------+
| Type   | Name                        | Status      |
+--------+-----------------------------+-------------+
| InnoDB | &table->autoinc_mutex       | os_waits=1  |
| InnoDB | &table->autoinc_mutex       | os_waits=1  |
| InnoDB | &table->autoinc_mutex       | os_waits=4  |
| InnoDB | &table->autoinc_mutex       | os_waits=1  |
| InnoDB | &table->autoinc_mutex       | os_waits=12 |
| InnoDB | &dict_sys->mutex            | os_waits=1  |
| InnoDB | &log_sys->mutex             | os_waits=12 |
| InnoDB | &fil_system->mutex          | os_waits=11 |
| InnoDB | &kernel_mutex               | os_waits=1  |
| InnoDB | &dict_table_stats_latches[i] | os_waits=2  |
| InnoDB | &dict_table_stats_latches[i] | os_waits=54 |
| InnoDB | &dict_table_stats_latches[i] | os_waits=1  |
| InnoDB | &dict_table_stats_latches[i] | os_waits=31 |
| InnoDB | &dict_table_stats_latches[i] | os_waits=41 |
| InnoDB | &dict_table_stats_latches[i] | os_waits=12 |
| InnoDB | &dict_table_stats_latches[i] | os_waits=1  |
| InnoDB | &dict_table_stats_latches[i] | os_waits=90 |
| InnoDB | &dict_table_stats_latches[i] | os_waits=1  |
| InnoDB | &dict_operation_lock         | os_waits=13 |
| InnoDB | &log_sys->checkpoint_lock    | os_waits=66 |
| InnoDB | combined &block->lock        | os_waits=2  |
+--------+-----------------------------+-------------+
```

Проанализировав распечатку, на основе количества ожиданий можно понять, какие части InnoDB являются узкими местами. Всюду, где есть мьютекс, существует опасность конкуренции. Возможно, следует написать скрипт для агрегирования распечатки, которая может быть очень большой.

Существует три основных способа избежать узких мест: держаться подальше от известных слабостей InnoDB, ограничить степень конкурентности и постараться найти

компромисс между активным ожиданием, нагружающим процессор, и ожиданием на уровне ОС, потребляющим ресурсы. Эти вопросы обсуждались ранее — в этом приложении и в главе 8.

Состояние репликации

В MySQL есть несколько команд для мониторинга репликации. Выполнение команды `SHOW MASTER STATUS` на главном сервере показывает его состояние и конфигурацию репликации:

```
mysql> SHOW MASTER STATUS\G
*************************** 1. row ***************************
            File: mysql-bin.000079
        Position: 13847
    Binlog_Do_DB:
Binlog_Ignore_DB:
```

Выводится текущая позиция в двоичном журнале главного сервера. Получить список всех двоичных журналов позволит команда `SHOW BINARY LOGS`:

```
mysql> SHOW BINARY LOGS
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| mysql-bin.000044 |     13677 |
...
| mysql-bin.000079 |     13847 |
+------------------+-----------+
36 rows in set (0.18 sec)
```

Чтобы просмотреть события в двоичных журналах, выполните команду `SHOW BINLOG EVENTS`. В MySQL 5.5 также можно использовать команду `SHOW RELAYLOG EVENTS`.

Получить состояние и конфигурацию подчиненного сервера позволяет команда `SHOW SLAVE STATUS`. Мы не стали включать в текст выдаваемую ею распечатку, потому что она довольно длинная, но сделаем на ее счет несколько замечаний. Во-первых, в ней показано состояние потока ввода/вывода и всех потоков SQL на подчиненном сервере, включая любые ошибки. Кроме того, видно, насколько далеко подчиненный сервер отстал от главного. И наконец, для резервного копирования и клонирования подчиненных серверов печатаются три набора координат в двоичном журнале:

- `Master_Log_File/Read_Master_Log_Pos` — позиция, с которой поток ввода/вывода читает двоичные журналы главного сервера;
- `Relay_Log_File/Relay_Log_Pos` — позиция команды в журнале ретрансляции подчиненного сервера, которую исполняет поток SQL;
- `Relay_Master_Log_File/Exec_Master_Log_Pos` — позиция команды в двоичном журнале главного сервера, которую исполняет поток SQL. Логически это та же позиция, которая обозначается парой `Relay_Log_File/Relay_Log_Pos`, но только в журналах главного, а не подчиненного сервера. Иными словами, в указанной позиции в обоих журналах находится одно и то же событие.

База данных INFORMATION_SCHEMA

База данных INFORMATION_SCHEMA представляет собой набор системных представлений и соответствует стандарту SQL. В MySQL реализовано значительное число описанных в стандарте представлений, а также добавлены свои. В версии MySQL 5.1 многим представлениям соответствуют команды SHOW, например SHOW FULL PROCESSLIST и SHOW STATUS. Но есть и такие, для которых соответствующих команд SHOW не существует.

Прелесть представлений из базы данных INFORMATION_SCHEMA заключается в том, что их можно опрашивать с помощью стандартных команд SQL. В результате достигается куда большая гибкость, чем при использовании команд SHOW, порождающих результирующие наборы, которые невозможно агрегировать, соединять и которыми нельзя как-то манипулировать с использованием обычных средств SQL. Наличие данных в системных представлениях дает возможность составлять очень интересные и полезные запросы.

Например, какие таблицы в демонстрационной базе данных Sakila ссылаются на таблицу actor? При наличии последовательного соглашения об именовании определить это совсем не трудно:

```
mysql> SELECT TABLE_NAME FROM INFORMATION_SCHEMA.COLUMNS
    -> WHERE TABLE_SCHEMA='sakila' AND COLUMN_NAME='actor_id'
    -> AND TABLE_NAME <> 'actor';
+------------+
| TABLE_NAME |
+------------+
| actor_info |
| film_actor |
+------------+
```

Для нескольких примеров из этой книги нужно было найти индексы по нескольким столбцам. Вот соответствующий запрос:

```
mysql>  SELECT TABLE_NAME, GROUP_CONCAT(COLUMN_NAME)
    ->  FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
    ->  WHERE TABLE_SCHEMA='sakila'
    ->  GROUP BY TABLE_NAME, CONSTRAINT_NAME
    ->  HAVING COUNT(*) > 1;
+---------------+-------------------------------------+
| TABLE_NAME    | GROUP_CONCAT(COLUMN_NAME)           |
+---------------+-------------------------------------+
| film_actor    | actor_id,film_id                    |
| film_category | film_id,category_id                 |
| rental        | customer_id,rental_date,inventory_id |
+---------------+-------------------------------------+
```

Можно писать и более сложные запросы, точно так же, как для обычных таблиц. Сайт MySQL Forge (http://forge.mysql.com) — отличное место для поиска запросов к этим представлениям; там же вы можете поделиться своими идеями. Имеются примеры запросов для поиска дублирующих или избыточных индексов, поиска индексов с очень низкой избирательностью и многое, многое другое. Множество полезных представлений, написанных поверх представлений INFORMATION_SCHEMA,

имеется также в проекте common_schema Шломи Ноача (http://code.openark.org/forge/common_schema).

Основной недостаток представлений заключается в том, что некоторые из них работают очень медленно по сравнению с соответствующими командами SHOW. Как правило, они отбирают все данные, сохраняют их во временной таблице, а затем делают эту таблицу доступной для запроса. Запрос к таблицам INFORMATION_SCHEMA на сервере с большим количеством данных или множеством таблиц может дать большую нагрузку на сервер. Эта нагрузка может привести к тому, что он остановится или перестанет отвечать на запросы других пользователей, поэтому будьте осторожны при использовании таких запросов на сильно загруженных промышленных серверах. Основными таблицами, запросы к которым могут быть потенциально опасными, являются те, в которых содержатся метаданные: TABLES, COLUMNS, REFERENTIAL_CONSTRAINTS, KEY_COLUMN_USAGE и т. д. Запросы к этим таблицам могут вынудить MySQL запрашивать подсистему хранения о таких данных, как индексная статистика в таблицах на сервере, что особенно проблематично в InnoDB.

В текущей версии эти представления необновляемы. Хотя они позволяют извлечь параметры сервера, обновить их так, чтобы изменилось его поведение, не удастся. На практике эти ограничения означают, что для конфигурирования сервера попрежнему необходимо пользоваться командами SHOW и SET, пусть даже представления из базы INFORMATION_SCHEMA очень удобны для других целей.

Таблицы Inno DB

В MySQL 5.1 и более новых версиях плагин InnoDB создает несколько таблиц в INFORMATION _SCHEMA. Это очень полезно. В MySQL 5.5 их больше, а в пока не выпущенной версии MySQL 5.6 — еще больше. В MySQL 5.1 существуют следующие таблицы:

- INNODB_CMP и INNODB_CMP_RESET. В этих таблицах представлена информация о данных, сжатых в новом файловом формате InnoDB Barracuda. Во второй таблице показана та же информация, что и в первой, но у нее имеется побочный эффект сброса содержащихся в ней данных, примерно как в команде FLUSH;
- INNODB_CMPMEM и INNODB_CMPMEM_RESET. В этих таблицах показана информация о страницах буферного пула, используемых для сжатых данных InnoDB. Вторая таблица — это снова таблица со сбросом;
- INNODB_TRX и INNODB_LOCKS. Эти таблицы показывают обычные транзакции и транзакции, которые удерживают блокировки и ждут их. Эти таблицы очень важны для диагностики проблем ожидания блокировки и длительных транзакций.

Помимо этих таблиц, в MySQL 5.5 добавлена таблица INNODB_LOCK_WAITS, облегчающая диагностику других типов проблем с ожиданием блокировки. В MySQL 5.6 будут добавлены таблицы, содержащие больше информации о внутренних функциях InnoDB, включая буферный пул и словарь данных, а также таблица под названием INNODB_METRICS, которая станет альтернативой использованию Performance Schema.

Таблицы в Percona Server

В Percona Server в базу данных `INFORMATION_SCHEMA` добавлено множество таблиц. В версии MySQL 5.5 у сервера 39 таблиц, а в Percona Server 5.5 — уже 61 таблица. Приведем обзор дополнительных таблиц.

- **Таблицы «Статистика пользователя».** Они возникли в патчах Google для MySQL. В них представлены показатели активности для клиентов, индексов, таблиц, потоков и пользователей. Мы упоминали об их использовании в этой книге, например, в случае, если репликация приближается к пределу своей возможности идти в ногу с главным сервером.
- **Словарь данных InnoDB.** Серия таблиц, в которых представлен внутренний словарь данных InnoDB в виде таблиц только для чтения: столбцы, внешние ключи, индексы, статистика и т. д. Они очень полезны для изучения и понимания подхода к базе данных, реализованного в InnoDB, который может отличаться от подхода MySQL. Дело в том, что работа MySQL зависит от .frm-файлов для хранения словаря данных. Аналогичные таблицы будут включены в MySQL 5.6, когда она будет выпущена.
- **Буферный пул InnoDB.** Эти таблицы позволяют вам запрашивать буферный пул, как если бы это была таблица, в которой каждая страница является строкой. Таким образом, вы можете видеть, какие страницы находятся в буферном пуле, к каким типам они относятся и т. д. Эти таблицы доказали свою полезность при диагностике таких проблем, как раздутый буфер вставки.
- **Временные таблицы.** В этих таблицах представлен тот же тип информации, что и в таблице `INFORMATION_SCHEMA.TABLES`, но для временных таблиц. Одна таблица предназначена для временных таблиц вашего сеанса и одна — для всех временных таблиц всего сервера. Обе весьма полезны для того, чтобы понять, какие временные таблицы существуют, в каких сеансах и сколько места они занимают.
- **Разные таблицы.** Существует парочка других таблиц, которые позволяют получить данные о времени выполнения запросов, файлах, табличном пространстве и другую информацию о внутренних процессах InnoDB.

Документация о дополнительных таблицах Percona Server доступна на сайте http://www.percona.com/doc/.

Performance Schema

Performance Schema (она находится в базе данных `PERFORMANCE_SCHEMA`) — это, начиная с MySQL 5.5, новый дом для улучшенных инструментов MySQL. Мы частично касались ее в главе 3.

По умолчанию Performance Schema отключена, и нужно включить как ее, так и особые точки оснащения инструментами (потребителей), которые вы хотите собирать. Мы провели эталонное тестирование сервера в нескольких разных конфигурациях и обнаружили, что Performance Schema вносит в общий объем издержек от 8 до 11 %, даже не собирая данные, и от 19 до 25 % — при всех включенных потре-

бителях в зависимости от того, заключалась ли рабочая нагрузка только в чтении или и в чтении и записи. Насколько велики эти затраты, решайте сами.

Эту проблему собираются решить в MySQL 5.6, особенно когда сама функция включена, но все точки оснащения инструментами отключены. Тем самым использование Performance Schema станет более приемлемым для некоторых пользователей: ее будут включать, но оставлять неактивной, пока не потребуется собирать каких-либо данные.

В MySQL 5.5 Performance Schema содержит таблицы, которые оснащают инструментами экземпляры условных переменных, мьютексов, блокировок чтения/записи и файлового ввода/вывода. Кроме того, существуют таблицы, в которых оснащаются инструментами периоды ожидания в экземплярах. Запросы к этим таблицам с соединениями с таблицами экземпляров должны интересовать вас в первую очередь. Эти таблицы ожидания событий существуют в нескольких вариациях, содержащих текущую и давно полученную информацию о поведении и производительности сервера. Наконец, существует группа таблиц настройки, которая используется для включения или отключения конкретных потребителей.

В версии MySQL 6 количество таблиц в Performance Schema увеличено с 17 до 49. Это означает, что теперь в MySQL используется намного больше инструментов! В частности, оснащаются инструментами операторы SQL, этапы операторов (в основном то же самое, что и состояние потока, которое вы можете увидеть в с помощью команды `SHOW PROCESS LIST`), таблицы, индексы, хосты, потоки, пользователи, аккаунты и множество разнообразных сводных таблиц и таблиц истории.

Как же можно использовать эти таблицы? Их уже 49, пора бы уже кому-нибудь написать инструменты для их использования. О хороших примерах старомодного SQL-запроса к таблицам Performance Schema можно прочитать в постах в блоге инженера Oracle Марка Лейта (Mark Leith), таких как http://www.markleith.co.uk/?p=471.

Итоги

Основными средствами MySQL для изучения внутренних процессов сервера являются команды `SHOW`, однако ситуация меняется. Введение в MySQL 5.1 подключаемых таблиц `INFORMATION_SCHEMA` позволило плагину InnoDB добавить несколько очень полезных инструментальных средств. Percona Server эти возможности расширил еще больше. Тем не менее способность читать и интерпретировать распечатку, которую выдает команда `SHOW ENGINE INNODB STATUS`, остается важным навыком для управления InnoDB. В MySQL 5.5 и более поздних версиях появилась Performance Schema — возможно, самое мощное и полное средство для проверки внутренних процессов сервера. Самое замечательное в Performance Schema то, что она основана на времени в том смысле, что MySQL наконец получила инструменты, оценивающие истекшее время, а не просто подсчитывающие операции.

Приложение В. Передача больших файлов

Копирование, упаковка и распаковка очень больших файлов (часто по сети) — задачи, очень часто встречающиеся при администрировании MySQL, инициализации серверов, клонировании подчиненных серверов, выполнении резервного копирования и восстановления. Не всегда самый очевидный способ выполнения таких операций является наиболее быстрым и лучшим, а разница между хорошим и плохим методами может оказаться весьма значительной. В данном приложении мы покажем несколько примеров применения стандартных утилит UNIX для передачи большого файла, содержащего образ резервной копии, с одного сервера на другой.

Обычно начинают с неупакованного файла, например табличного пространства InnoDB или файлов журнала. Кроме того, желательно, чтобы по завершении копирования файл в месте назначения тоже был распакован. Другая распространенная ситуация — начать с упакованного файла, например образа резервной копии, и в результате получить распакованный файл.

Если пропускная способность сети ограниченна, то обычно имеет смысл посылать файл по сети в сжатом виде. Кроме того, во избежание компрометации данных может понадобиться защитить файл во время передачи — это требование типично для образов резервных копий.

Копирование файлов

Таким образом, задача заключается в том, чтобы эффективно выполнить следующие операции.

1. Упаковать данные (не обязательно).
2. Отправить их на другой компьютер.
3. Распаковать архив в месте назначения.
4. Проверить, не повреждены ли файлы в процессе копирования.

Мы провели эталонное тестирование разных методов достижения этих целей. Далее опишем наши действия и сделаем вывод о самом быстром способе.

При решении многих обсуждавшихся в этой книге задач, например при резервном копировании, следует решить, на какой машине выполнять сжатие. Если пропускная способность сети высока, то можно переносить образы резервных копий без сжатия и сэкономить ресурсы процессора на сервере MySQL для обработки запросов.

Наивный пример

Начнем с наивного примера. Требуется безопасно передать неупакованный файл с одной машины на другую, упаковать его по пути, а затем распаковать в месте назначения. На исходном сервере, назовем его `server1`, выполним следующие команды:

```
server1$ gzip -c /backup/mydb/mytable.MYD > mytable.MYD.gz
server1$ scp mytable.MYD.gz root@server2:/var/lib/myql/mydb/
```

А затем на конечном сервере `server2` дадим следующую команду:

```
server2$ gunzip /var/lib/mysql/mydb/mytable.MYD.gz
```

Наверное, это самый простой подход, но вовсе не самый эффективный, потому что все три шага — упаковка, копирование и распаковка — выполняются последовательно. На каждом шаге необходимо производить медленные операции дискового чтения-записи. Вот что на самом деле происходит при работе каждой из вышеупомянутых команд: `gzip` производит чтение и запись на сервере `server1`, `scp` читает с `server1` и записывает на `server2`, а `gunzip` читает и записывает на `server2`.

Метод с одним шагом

Более эффективно упаковывать и копировать файл на одном конце и распаковывать на другом за один шаг. В этот раз мы воспользуемся безопасным протоколом SSH, на котором основана программа SCP. Вот какую команду мы выполним на сервере `server1`:

```
server1$ gzip -c /backup/mydb/mytable.MYD | ssh root@server2"gunzip -c – > /var/lib
>/mysql/mydb/mytable.MYD"
```

Обычно этот метод работает гораздо быстрее, чем предыдущий, потому что существенно снижается объем дискового ввода/вывода: все сводится к чтению на `server1` и записи на `server2`. Поэтому диск может выполнять операции последовательно.

Можно также воспользоваться механизмом сжатия, встроенным в SSH, но мы показали, как упаковывать и распаковывать с помощью конвейера, поскольку это более гибкий метод. Например, если распаковывать файл на другом конце не нужно, то сжатие на уровне SSH не подойдет.

Этот метод можно улучшить, если немного поиграть с флагами. Так, при задании флага `-1` программа gzip пакует быстрее. Обычно коэффициент сжатия при этом уменьшается несущественно, зато скорость возрастает очень заметно, а это важно. Можно применять и другие алгоритмы компрессии. Например, если нужна очень высокая степень сжатия и неважно, сколько это займет времени, то вместо gzip можно использовать bzip2. Если же требуется максимально быстрая компрессия, то лучше выбрать какой-нибудь архиватор на основе алгоритма LZO. Размер сжатого файла может оказаться процентов на 20 больше, зато паковаться он будет примерно в пять раз быстрее.

Устранение затрат на шифрование

SSH — не самый быстрый способ транспортировки данных по сети, поскольку ему присущи затраты на шифрование и дешифрование. Если шифровать данные не требуется, то можно просто побитово скопировать их по сети с помощью программы netcat. Для неинтерактивных операций, которые нас и интересуют, она вызывается просто как nc.

Рассмотрим пример. Начнем прослушивать порт 12345 (подойдет любой неиспользуемый порт) на сервере server2 и распаковывать все, что поступает через него в файл с требуемым именем:

```
server2$ nc -l -p 12345 | gunzip -c - > /var/lib/mysql/mydb/mytable.MYD
```

На сервере server1 запустим еще один экземпляр netcat, который будет отправлять данные в порт, прослушиваемый в месте назначения. Флаг `-q` говорит netcat, что после обнаружения конца входного файла соединение нужно закрыть. Это, в свою очередь, приведет к тому, что netcat на принимающей стороне закроет полученный файл и завершится:

```
server1$ gzip -c - /var/lib/mysql/mydb/mytable.MYD | nc -q 1 server2 12345
```

Еще проще воспользоваться программой tar, при этом по сети будут передаваться также имена файлов. Это устраняет потенциальный источник ошибок, так как файлы автоматически записываются в нужное место. Параметр z заставляет tar использовать gzip для сжатия и распаковки. Вот какую команду следует выполнить на сервере server2:

```
server2$ nc -l -p 12345 | tar xvzf -
```

А вот такую — на server1:

```
server1$ tar cvzf - /var/lib/mysql/mydb/mytable.MYD | nc -q 1 server2 12345
```

Эти команды можно поместить в один скрипт, который будет эффективно сжимать и копировать файлы по сети, а затем распаковывать их на другой стороне.

Другие способы

Еще один вариант — воспользоваться программой rsync. Она удобна тем, что позволяет легко поддерживать зеркальное соответствие между исходным и конечным серверами и возобновлять прерванную передачу. Но если используемый в ней алгоритм вычисления разности между двоичными файлами невозможно эффективно применить, то она работает не очень хорошо. Пожалуй, имеет смысл применять ее в тех случаях, когда изначально известно, что большую часть файла посылать не потребуется, например для завершения прерванной операции копирования с помощью nc.

Экспериментировать с передачей файлов следует в спокойных условиях, когда нет никакой спешки, так как для нахождения самого быстрого варианта придется

действовать методом проб и ошибок. Какой способ будет работать быстрее всего, зависит от конкретной системы. Основными факторами тут являются количество дисков, сетевых карт и процессоров, а также их относительное быстродействие. Мы рекомендуем для начала выполнить команду `vmstat -n 5`, чтобы узнать, что является узким местом — диски или процессор.

Если часть процессоров простаивает, то, возможно, операцию удастся ускорить, запустив несколько процедур копирования параллельно. И наоборот, если процессор является узким местом, зато у дисков и сети имеется свободная пропускная способность, то можно исключить сжатие. И не забывайте вести мониторинг производительности серверов, чтобы понять, располагают ли они резервной пропускной способностью. Попытка запустить слишком много параллельных операций может даже привести к замедлению работы.

Эталонные тесты копирования файлов

Для сравнения в табл. В.1 показано, за какое время нам удалось скопировать тестовый файл по стандартной локальной сети Ethernet со скоростью 100 Мбит/с. Размер исходного файла составлял 738 Мбайт, а упакованного программой gzip с флагами по умолчанию — 100 Мбайт. На исходной и конечной машинах было достаточно памяти, ресурсов процессора и свободного места на диске, узким местом была сеть.

Таблица В.1. Эталонные тесты копирования файлов по сети

Метод	Время, с
rsync без сжатия	71
scp без сжатия	68
nc без сжатия	67
rsync со сжатием (-z)	63
gzip, scp и gunzip	60 (44 + 10 + 6)
ssh со сжатием	44
nc со сжатием	42

Обратите внимание на то, как сжатие ускоряет передачу файла по сети: три метода, в которых сжатие не применялось, оказались самыми медленными. Но, конечно, ваши результаты могут отличаться от полученных нами. Если процессоры и диски не отличаются быстродействием, зато имеется гигабитная сеть Ethernet, то узким местом будут чтение и компрессия данных, поэтому, может быть, лучше обойтись без сжатия.

Кстати, часто бывает гораздо выгоднее использовать быструю архивацию, например `gzip --fast`, чем стандартные уровни компрессии, которые потребляют много процессорного времени, давая лишь небольшой выигрыш в коэффициенте сжатия. Мы в своем тесте довольствовались уровнем сжатия по умолчанию.

Последний шаг — проверка того, что данные не были искажены в процессе передачи данных. Тут в вашем распоряжении много разнообразных методов, например программа md5sum, применение которой, правда, довольно затратно, так как она должна еще раз прочитать весь файл. Тут проявляется еще одна грань пользы сжатия: алгоритм компрессии, как правило, уже включает по меньшей мере циклический избыточный код (CRC), который отлавливает все ошибки, так что контроль отсутствия искажений вы получаете бесплатно.

Приложение Г. Команда EXPLAIN

В этом приложении описывается, как с помощью команды EXPLAIN получать сведения о плане выполнения запроса и как интерпретировать результаты этой операции. Команда EXPLAIN — основной способ узнать, какие решения принимает оптимизатор запросов. У нее есть много ограничений, и не всегда она говорит правду, но поскольку ничего лучшего все равно не существует, то имеет смысл научиться ею пользоваться, тогда вы сможете делать обоснованные предположения о том, как выполняются запросы.

Вызов команды EXPLAIN

Чтобы воспользоваться командой EXPLAIN, достаточно в запросе поставить слово EXPLAIN перед словом SELECT. MySQL пометит этот запрос специальным флагом. Во время его обработки этот флаг заставит сервер сообщать информацию о каждом шаге плана выполнения, а не исполнять его. При этом возвращается одна или несколько строк, описывающих этапы плана выполнения и порядок их запуска.

Далее показан простейший из возможных результатов применения EXPLAIN:

```
mysql> EXPLAIN SELECT 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra: No tables used
```

Для каждой встречающейся в запросе таблицы выводится одна строка. Если соединяются две таблицы, то будут выведены две строки. Если одна и та же таблица попадается дважды, например при соединении таблицы с самой собой, строк тоже будет две. В этом контексте семантика слова «таблица» довольно широка: оно может обозначать подзапрос, результат объединения (UNION) и т. д. Позже мы увидим, почему так происходит.

Существует две важные разновидности команды EXPLAIN.

- Команда EXPLAIN EXTENDED, по всей видимости, ведет себя так же, как обычная EXPLAIN, но говорит серверу, что требуется произвести обратную компиляцию плана выполнения в команду SELECT. Чтобы просмотреть сгенерированный за-

прос, следует сразу после завершения `EXPLAIN EXTENDED` выполнить команду `SHOW WARNINGS`. Отображаемая при этом директива получена непосредственно из плана выполнения, а не из исходной SQL-команды, которая к этому моменту уже преобразована в некую структуру данных. В большинстве случаев исходная и восстановленная команды будут различаться. Сравнив их, вы поймете, как оптимизатор трансформировал ваш запрос. Команда `EXPLAIN EXTENDED` появилась в версии MySQL 5.0, а в версии MySQL 5.1 был добавлен дополнительный столбец `filtered` (подробнее об этом поговорим позже).

- Команда `EXPLAIN PARTITIONS` показывает, к каким секциям обращается запрос (если это имеет смысл). Она доступна начиная с версии MySQL 5.1.

Часто считают, что при наличии слова `EXPLAIN` MySQL не выполняет запрос. На самом деле, если запрос содержит подзапрос в разделе `FROM`, MySQL обрабатывает этот подзапрос, помещает результат во временную таблицу, после чего продолжает оптимизацию внешнего запроса. Сервер обязан выполнить все подзапросы перед тем, как сможет завершить оптимизацию внешнего запроса, а в этом и состоит суть `EXPLAIN`[1].

Таким образом, если запрос содержит сложные подзапросы или представления, в которых применяется алгоритм `TEMPTABLE`, то выполнение `EXPLAIN` может означать большой объем работы для сервера.

Имейте в виду, что команда `EXPLAIN` не более чем аппроксимация, иногда хорошая, а иногда очень далекая от истины. Приведем перечень ее ограничений.

- `EXPLAIN` ничего не говорит о том, как влияют на запрос триггеры, хранимые функции и функции, определенные пользователем.
- Она не работает с хранимыми процедурами, хотя можно разложить процедуру на отдельные запросы и вызвать `EXPLAIN` для каждого из них.
- Она ничего не говорит об оптимизациях, которые MySQL производит уже на этапе выполнения запроса.
- Часть отображаемой статистической информации — всего лишь оценка, иногда очень неточная.
- Она не показывает все, что можно было бы сообщить о плане выполнения запроса (когда представляется возможность, разработчики MySQL включают дополнительную информацию).
- Она не делает различий между некоторыми операциями, называя их одинаково. Например, словом filesort обозначается и сортировка в памяти, и сортировка с помощью временных файлов, а при использовании временных таблиц — все равно, на диске или в памяти — она сообщает: `Using temporary`.
- Результат может сбить с толку. Например, она может сообщить о полном сканировании индекса для запроса, который отбирает небольшое число строк благодаря наличию ключевого слова `LIMIT`. (В версии MySQL 5.1 команда `EXPLAIN` выдает более точную информацию о количестве подлежащих просмотру строк, но в предыдущих версиях ключевое слово `LIMIT` не принималось во внимание.)

[1] Это ограничение будет устранено в MySQL 5.6.

Переписывание запросов, отличных от SELECT. Команда EXPLAIN работает только для запросов типа SELECT, но не для хранимых процедур, команд INSERT, UPDATE, DELETE и пр. Однако некоторые отличные от SELECT запросы можно переписать в виде, который можно объяснить. Для этого следует преобразовать команду в эквивалентный запрос SELECT, который выбирает те же самые столбцы. Любой упоминаемый в исходной директиве столбец должен встречаться либо в списке SELECT, либо в условии соединения, либо в разделе WHERE.

Предположим, например, что нужно переписать следующую команду UPDATE, сделав ее объясняемой:

```
UPDATE sakila.actor
    INNER JOIN sakila.film_actor USING (actor_id)
SET actor.last_update=film_actor.last_update;
```

Следующая директива EXPLAIN не эквивалентна этой команде UPDATE, так как от сервера не требуется выбирать столбец last_update ни из одной таблицы:

```
mysql> EXPLAIN SELECT film_actor.actor_id
    -> FROM sakila.actor
    ->    INNER JOIN sakila.film_actor USING (actor_id)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: index
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: NULL
         rows: 200
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.actor.actor_id
         rows: 13
        Extra: Using index
```

Это различие очень важно. EXPLAIN показывает, что MySQL будет использовать покрывающие индексы, а это было бы невозможно в случае выборки и обновления столбца last_update. Следующий вариант гораздо ближе к оригиналу:

```
mysql> EXPLAIN SELECT film_actor.last_update, actor.last_update
    -> FROM sakila.actor
    ->    INNER JOIN sakila.film_actor USING (actor_id)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
```

```
           type: ALL
 possible_keys: PRIMARY
           key: NULL
       key_len: NULL
           ref: NULL
          rows: 200
         Extra:
*************************** 2. row ***************************
            id: 1
  select_type: SIMPLE
         table: film_actor
          type: ref
 possible_keys: PRIMARY
           key: PRIMARY
       key_len: 2
           ref: sakila.actor.actor_id
          rows: 13
         Extra:
```

Переписывание запросов — не точная наука, но часто этого бывает достаточно, чтобы узнать, как будет выполняться запрос[1].

Важно понимать, что не существует такого понятия, как эквивалентный запрос на чтение, который показал бы план выполнения запроса на запись. От запроса типа `SELECT` требуется найти и вернуть всего одну копию данных. В то же время запрос, модифицирующий данные, должен найти и изменить все копии данных во всех индексах. Часто это оказывается гораздо затратнее, чем эквивалентный запрос `SELECT`.

Столбцы результата команды EXPLAIN

Результат команды `EXPLAIN` всегда состоит из одних и тех же столбцов (за исключением команды `EXPLAIN EXTENDED`, которая в версии MySQL 5.1 добавляет столбец `filtered`, и команды `EXPLAIN PARTITIONS`, которая добавляет столбец `partitions`). Изменяются лишь количество и содержимое строк. Однако, чтобы не усложнять примеры, мы не всегда будем показывать все столбцы.

В следующих разделах объясним назначение столбцов в результате, выдаваемом командой `EXPLAIN`. Имейте в виду, что порядок строк такой, в котором MySQL фактически выполняет части запроса, а он не всегда совпадает с порядком упоминания в исходной SQL-команде.

Столбец id

В этом столбце всегда находится число, идентифицирующее запрос `SELECT`, которому принадлежит строка. Если в исходной команде нет ни подзапросов, ни ключевого слова `UNION`, то существует лишь один запрос `SELECT`, так что во всех строках этот

[1] В MySQL 5.6 будет возможность использовать команду EXPLAIN для запросов без SELECT. Ура!

столбец будет содержать 1. В противном случае внутренние запросы SELECT обычно нумеруются последовательно в том порядке, в котором стоят в исходной команде.

MySQL разбивает все запросы SELECT на простые и составные, причем составные запросы можно отнести к одной из трех категорий: простые подзапросы, так называемые производные таблицы (подзапросы в разделе FROM) и объединения UNION[1]. Вот пример простого подзапроса:

```
mysql> EXPLAIN SELECT (SELECT 1 FROM sakila.actor LIMIT 1) FROM sakila.film;
 +----+-------------+-------+...
 | id | select_type | table |...
 +----+-------------+-------+...
 |  1 | PRIMARY     | film  |...
 |  2 | SUBQUERY    | actor |...
 +----+-------------+-------+...
```

Подзапросы в разделе FROM и UNION усложняют содержимое столбца id. Вот пример несложного подзапроса в разделе FROM:

```
mysql> EXPLAIN SELECT film_id FROM (SELECT film_id FROM sakila.film) AS der;
 +----+-------------+------------+...
 | id | select_type | table      |...
 +----+-------------+------------+...
 |  1 | PRIMARY     | <derived2> |...
 |  2 | DERIVED     | film       |...
 +----+-------------+------------+...
```

Как вы знаете, этот запрос выполняется с помощью анонимной временной таблицы. MySQL ссылается на данную временную таблицу из внешнего запроса по псевдониму der, который в более сложных запросах указывается в столбце ref.

Наконец, приведем запрос с UNION:

```
mysql> EXPLAIN SELECT 1 UNION ALL SELECT 1;
 +------+--------------+------------+...
 | id   | select_type  | table      |...
 +------+--------------+------------+...
 |  1   | PRIMARY      | NULL       |...
 |  2   | UNION        | NULL       |...
 | NULL | UNION RESULT | <union1,2> |...
 +------+--------------+------------+...
```

Обратите внимание на дополнительную строку, соответствующую результату выполнения UNION. Итоги обработки UNION всегда помещаются в анонимную временную таблицу, из которой MySQL затем считывает их обратно. Эта временная таблица отсутствует в исходной SQL-команде, поэтому в столбце id для нее стоит NULL. В отличие от предыдущего примера, иллюстрирующего подзапрос в разделе FROM,

[1] Утверждение «всякий подзапрос в разделе FROM является производной таблицей» истинно, однако утверждение «всякая производная таблица является подзапросом в разделе FROM» ложно. В языке SQL понятие «производная таблица» употребляется более широко.

временная таблица, порождаемая в ходе выполнения запроса, показана в последней, а не в первой строке.

До сих пор все было довольно просто, но комбинация всех трех категорий команд может существенно усложнить результат, в чем мы скоро убедимся.

Столбец select_type

Этот столбец показывает, какому запросу SELECT соответствует строка, простому или составному, и если составному, то к какой из трех категорий он относится. Атрибут SIMPLE означает, что запрос не содержит ни подзапросов, ни UNION. Если же в запросе имеются такие компоненты, то самый внешний запрос помечается признаком PRIMARY, а остальные — следующим образом:

- SUBQUERY. Запрос SELECT, который содержится в подзапросе, находящемся в разделе SELECT (иными словами, не в разделе FROM), помечается признаком SUBQUERY;
- DERIVED. Значение DERIVED означает, что данный запрос SELECT является подзапросом в разделе FROM. MySQL выполняет его рекурсивно и помещает результат во временную таблицу. Внутри сервер ссылается на нее по имени «производная таблица», так как она произведена из подзапроса;
- UNION. Второй и последующий запросы SELECT, входящие в объединение UNION, помечаются признаком UNION. При этом первый SELECT помечается так, будто он является частью внешнего запроса. Именно поэтому первый SELECT в UNION из предыдущего примера помечен как PRIMARY. Если бы это объединение UNION было частью подзапроса в разделе FROM, то его первый SELECT был бы помечен как DERIVED;
- UNION RESULT. Запрос SELECT, применяемый для выборки результатов из временной таблицы, созданной в ходе выполнения UNION, помечается признаком UNION RESULT.

Помимо этих значений, признаки SUBQUERY и UNION могут быть дополнительно квалифицированы как DEPENDENT и UNCACHEABLE. DEPENDENT означает, что результат SELECT зависит от данных, встречающихся во внешнем запросе, UNCACHEABLE говорит о том, что нечто в запросе SELECT мешает поместить результаты в кэш Item_cache. (Кэш Item_cache не задокументирован — это не то же самое, что кэш запросов, хотя помещению в тот и другой могут воспрепятствовать одни и те же конструкции, например функция RAND().)

Столбец table

Этот столбец показывает, к какой таблице относится данная строка. В большинстве случаев все просто — это таблица или ее псевдоним, встречающиеся в SQL-команде.

Читать этот столбец следует сверху вниз, чтобы видеть порядок соединения таблиц, выбранный оптимизатором. Например, в следующем примере MySQL выбрала порядок соединения, отличный от указанного в самом запросе:

```
mysql> EXPLAIN SELECT film.film_id
    -> FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    ->    INNER JOIN sakila.actor USING(actor_id);
+----+-------------+------------+...
| id | select_type | table      |...
+----+-------------+------------+...
|  1 | SIMPLE      | actor      |...
|  1 | SIMPLE      | film_actor |...
|  1 | SIMPLE      | film       |...
+----+-------------+------------+...
```

Вспомните диаграммы в виде деревьев, наклоненных влево, из главы 6. Планы выполнения в MySQL всегда представляются деревьями, наклоненными влево. Если положить такой план на бок, можно читать его по порядку названия листьев, и это будет соответствовать рядам в выводе команды `EXPLAIN`. План предыдущего запроса выглядит так, как показано на рис. Г.1.

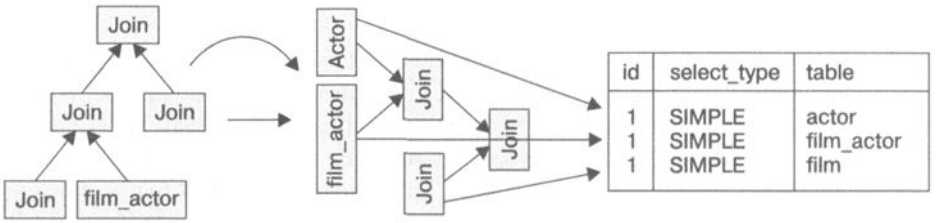


Рис. Г.1. Соответствие плана выполнения запроса строкам результата команды EXPLAIN

Производные таблицы и объединения. Столбец `table` становится существенно сложнее, если запрос содержит подзапрос в разделе `FROM` или объединение `UNION`. В этих случаях не существует таблицы, на которую можно было бы сослаться, так как анонимная временная таблица существует лишь до тех пор, пока запрос выполняется.

Для подзапроса в разделе `FROM` столбец `table` принимает вид `<derivedN>`, где *N* — идентификатор подзапроса. Это всегда опережающая ссылка, то есть *N* относится к строке, которая стоит в выданном `EXPLAIN` результате ниже.

Для запросов, содержащих `UNION`, строка `UNION RESULT` в столбце `table` содержит список идентификаторов запросов (`id`), для которых производится объединение. Это всегда обратная ссылка, поскольку строка с признаком `UNION RESULT` встречается после всех строк, относящихся к частям `UNION`. Если в списке более 20 идентификаторов, то столбец `table` может быть обрезан и всех значений вы не увидите. К счастью,

нетрудно догадаться, какие строки были в него включены, поскольку идентификатор первой из них всегда виден, а все, что находится между ней и строкой UNION RESULT, так или иначе включено.

Пример с различными типами составных запросов SELECT. Далее в качестве примера приведен бессмысленный запрос, который тем не менее в компактной форме демонстрирует некоторые типы составных запросов SELECT:

```
1  EXPLAIN
2  SELECT actor_id,
3     (SELECT 1 FROM sakila.film_actor WHERE film_actor.actor_id =
4        der_1.actor_id LIMIT 1)
5  FROM (
6     SELECT actor_id
7     FROM sakila.actor LIMIT 5
8  ) AS der_1
9  UNION ALL
10  SELECT film_id,
11     (SELECT @var1 FROM sakila.rental LIMIT 1)
12  FROM (
13     SELECT film_id,
14        (SELECT 1 FROM sakila.store LIMIT 1)
15     FROM sakila.film LIMIT 5
16  ) AS der_2;
```

Ключевые слова LIMIT включены в него только для удобства — на случай, если вы захотите выполнить этот запрос без EXPLAIN и посмотреть результаты его работы. Вот как выглядит результат EXPLAIN:

```
+------+---------------------+------------+...
| id   | select_type         | table      |...
+------+---------------------+------------+...
|  1   | PRIMARY             | <derived3> |...
|  3   | DERIVED             | actor      |...
|  2   | DEPENDENT SUBQUERY  | film_actor |...
|  4   | UNION               | <derived6> |...
|  6   | DERIVED             | film       |...
|  7   | SUBQUERY            | store      |...
|  5   | UNCACHEABLE SUBQUERY| rental     |...
| NULL | UNION RESULT        | <union1,4> |...
+------+---------------------+------------+...
```

Мы специально построили запрос так, чтобы каждая его часть обращалась к разным таблицам и было понятно, что куда попадает. И тем не менее разобраться в нем трудно! Начнем сверху.

- Первая строка является опережающей ссылкой на производную таблицу der_1, которая в запросе помечена как <derived3>. Она соответствует строке 2 исходной SQL-команды. Чтобы понять, какие строки результата относятся к командам SELECT, являющимся частью <derived3>, смотрим вперед...
- ...на вторую строку с идентификатором 3. Значение 3 объясняется тем, что строка относится к третьему по порядку запросу SELECT и помечена признаком DERIVED,

поскольку соответствующий подзапрос встречается в разделе FROM (строки 6 и 7 исходной SQL-команды).

- Третья строка имеет идентификатор 2 и соответствует строке 3 исходной SQL-команды. Отметим, что она идет после строки с большим id, то есть соответствующий ей подзапрос выполняется позже, как и должно быть. Она помечена признаком DEPENDENT SUBQUERY, что означает, что ее результат зависит от итогов обработки внешнего запроса (также называемого *коррелированным подзапросом*). В данном случае внешний запрос начинается в строке 2 и выбирает данные из таблицы der_1.
- Четвертая строка помечена признаком UNION, следовательно, соответствует второму или последующему подзапросу SELECT в UNION. В столбце table находится значение <derived6>, это означает, что данные выбираются из результатов подзапроса в разделе FROM и добавляются во временную таблицу для результатов UNION. Как и раньше, чтобы найти в результате EXPLAIN строки, в которых показан план выполнения этого подзапроса, нужно заглянуть вперед.
- Пятая строка соответствует подзапросу der_2, определенному в строках 13, 14 и 15. EXPLAIN ссылается на него по имени <derived6>.
- Шестая строка — это обычный подзапрос в списке SELECT таблицы <derived6>. Его id равен 7, что существенно...
- ...так как он больше 5 — идентификатора седьмой строки. Почему это так важно? Потому что очерчивает границы подзапроса <derived6>. Каждая выведенная EXPLAIN строка с признаком DERIVED является началом вложенной области видимости. Как только появляется строка с меньшим идентификатором (в данном случае 5 меньше 6), вложенная область видимости закрывается. Поэтому мы знаем, что седьмая строка — часть списка SELECT, в котором выбираются данные из <derived6>, то есть часть списка SELECT из четвертой строки (строка 11 в исходной SQL-команде). Этот пример довольно легко понять, даже не зная о правилах вложенных областей видимости, но так бывает не всегда. Стоит также отметить, что эта строка помечена признаком UNCACHEABLE SUBQUERY из-за пользовательской переменной.
- Наконец, последняя строка помечена признаком UNION RESULT. Она представляет этап считывания строк из временной таблицы, соответствующей UNION. Можете, если хотите, начать с этой строки и двигаться в обратном направлении: она возвращает объединение результатов, полученных в строках с идентификаторами 1 и 4, которые, в свою очередь, ссылаются на <derived3> и <derived6>.

Как видите, комбинация сложных подзапросов SELECT может порождать трудные для восприятия результаты EXPLAIN. Знание правил облегчает их чтение, но нет ничего лучше практики.

При изучении результатов EXPLAIN часто приходится заглядывать вперед и возвращаться назад. Возьмем, к примеру, первую строку результата. Глядя на нее, невозможно сказать, что она является частью UNION. Это становится ясно только после ознакомления с последней строкой.

Столбец type

В руководстве по MySQL сказано, что в этом столбце отражается тип соединения, но нам кажется, что следует говорить скорее о методе доступа, иными словами, о том, как MySQL решила искать строки в таблице. Перечислим наиболее важные методы доступа в порядке от наихудшего к наилучшему.

- `ALL`. Этот подход обычно называют сканированием таблицы. В общем случае речь идет о том, что MySQL должна просмотреть таблицу от начала до конца, чтобы найти нужную строку. (Существуют исключения, например запросы с ключевым словом `LIMIT` или запросы, для которых в столбце `Extra` отображается значение `Using distinct/not exists`.)
- `index`. То же, что сканирование таблицы, только MySQL просматривает ее в порядке, задаваемом индексом, а не в порядке следования строк. Основное преимущество заключается в том, что не требуется сортировка, недостаток же — высокие затраты на чтение всей таблицы в порядке, задаваемом индексом. Обычно это означает, что строки выбираются произвольным образом, что крайне накладно.

  Если в столбце `Extra` при этом находится значение `Using index`, значит, MySQL использует покрывающий индекс и просматривает только данные в индексе, не обращаясь к строкам. Это менее затратно, чем сканирование таблицы в задаваемом индексом порядке.
- `range`. Просмотр диапазона — это ограниченная форма сканирования индекса. Просмотр начинается в определенной точке индекса и возвращает строки в некотором диапазоне значений. Это лучше, чем полное сканирование, так как не приходится перебирать индекс целиком. Очевидные примеры просмотра диапазона — запросы с условиями `BETWEEN` или `>` в разделе `WHERE`.

  Когда MySQL использует индекс для поиска в списке значений, например при вычислении предиката `IN()` или условий, объединенных связкой `OR`, также применяется просмотр диапазона. Однако это совершенно разные методы доступа с иными характеристиками производительности. Дополнительную информацию см. во врезке «Что такое условие поиска по диапазону» в главе 5.

  К этому методу применимы те же самые соображения о затратах, что и к методу `index`.
- `ref`. Это доступ по индексу (иногда он называется поиском по индексу), в результате которого возвращаются строки, соответствующие единственному заданному значению. Но таких строк может быть несколько, поэтому поиск сочетается с просмотром. Данный тип доступа возможен лишь в случае существования неуникального индекса или неуникального префикса ключа в уникальном индексе. Он называется `ref`, потому что значения ключей в индексе сравниваются с некоторой справочной (reference) величиной. Эта величина может быть как константой, так и значением из предыдущей таблицы, если в запросе участвуют несколько таблиц.

  Одним из вариантов `ref` является тип доступа `ref_or_null`. В этом случае MySQL должна выполнить второй просмотр для поиска записей с ключом `NULL`.
- `eq_ref`. Это поиск по индексу в случае, когда MySQL точно знает, что будет возвращено не более одного значения. Такой метод доступа применяется, когда MySQL

решает использовать первичный ключ или уникальный индекс для сравнения на равенство с некоторой справочной величиной. Этот тип доступа MySQL умеет отлично оптимизировать, так как заранее известно, что не придется просматривать диапазоны отвечающих условию строк или искать дополнительные строки после того, как одна уже найдена.

- `const`, `system`. Эти типы доступа MySQL применяет, когда в процессе оптимизации какую-то часть запроса можно преобразовать в константу. Например, если вы выбираете первичный ключ строки и помещаете его в условие `WHERE`, то MySQL может преобразовать этот запрос в константу. Затем соответствующая таблица, по существу, удаляется из операции соединения.
- `NULL`. Этот метод означает, что MySQL сумела разрешить запрос на фазе оптимизации, так что в ходе выполнения вообще не потребуется обращаться к таблице или индексу. Например, для выборки минимального значения проиндексированного столбца достаточно просмотреть только индекс, не обращаясь к таблице.

Столбец possible_keys

Этот столбец показывает, какие индексы можно было бы задействовать для выполнения запроса, исходя из того, к каким столбцам производится обращение и какие операторы сравнения используются. Список создается на ранних этапах фазы оптимизации, поэтому на последующих стадиях может выясниться, что некоторые индексы бесполезны.

Столбец key

Этот столбец показывает, какой индекс MySQL решила использовать для оптимизации доступа к таблице. Если этот индекс отсутствует в столбце `possible_keys`, значит, MySQL выбрала его по какой-то другой причине. Например, покрывающий индекс может быть выбран даже в том случае, если раздела `WHERE` нет вообще.

Иными словами, столбец `possible_keys` показывает, какие индексы могли бы способствовать эффективному поиску строк, тогда как столбец `key` говорит о том, на каком индексе оптимизатор остановился, чтобы минимизировать затраты на запрос (о показателях стоимости запроса см. главу 6). Приведем пример:

```
mysql> EXPLAIN SELECT actor_id, film_id FROM sakila.film_actor\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: index
possible_keys: NULL
          key: idx_fk_film_id
      key_len: 2
          ref: NULL
         rows: 5143
        Extra: Using index
```

Столбец key_len

Этот столбец показывает, сколько байтов индекса использует MySQL. Если задействованы лишь некоторые индексированные столбцы, то, зная это значение, можно определить, какие именно. Напомним, что MySQL 5.5 и более старых версий может использовать только левый префикс индекса. Например, первичный ключ таблицы `sakila.film_actors` состоит из двух столбцов типа `SMALLINT`, а длина типа `SMALLINT` — 2 байта, поэтому каждая запись в индексе содержит 4 байта. Рассмотрим такой запрос:

```
mysql> EXPLAIN SELECT actor_id, film_id FROM sakila.film_actor WHERE actor_id=4;
 ...+------+---------------+---------+---------+...
 ...| type | possible_keys | key     | key_len |...
 ...+------+---------------+---------+---------+...
 ...| ref  | PRIMARY       | PRIMARY | 2       |...
 ...+------+---------------+---------+---------+...
```

Исходя из значения столбца `key_len`, можно сделать вывод, что для поиска по индексу задействован только первый столбец, `actor_id`. Определяя, какие столбцы используются, не забывайте о кодировке символьных столбцов:

```
mysql> CREATE TABLE t (
    ->    a char(3) NOT NULL,
    ->    b int(11) NOT NULL,
    ->    c char(1) NOT NULL,
    ->    PRIMARY KEY  (a,b,c)
    -> ) ENGINE=MyISAM DEFAULT CHARSET=utf8 ;
mysql> INSERT INTO t(a, b, c)
    ->    SELECT DISTINCT LEFT(TABLE_SCHEMA, 3), ORD(TABLE_NAME),
    ->       LEFT(COLUMN_NAME, 1)
    ->    FROM INFORMATION_SCHEMA.COLUMNS:
mysql> EXPLAIN SELECT a FROM t WHERE a='sak' AND b = 112;
 ...+------+---------------+---------+---------+...
 ...| type | possible_keys | key     | key_len |...
 ...+------+---------------+---------+---------+...
 ...| ref  | PRIMARY       | PRIMARY | 13      |...
 ...+------+---------------+---------+---------+...
```

Длина ключа 13 байт в этом запросе равна сумме длин столбцов `a` и `b`. Длина столбца `a` составляет три символа, для каждого из которых в кодировке UTF8 может понадобиться до 3 байт, а столбец `b` представляет собой четырехбайтовое целое число.

MySQL не всегда показывает, какая часть индекса реально используется. Например, для запроса с предикатом `LIKE`, в котором производится сопоставление с шаблоном по совпадению префикса, MySQL сообщит, что задействована вся ширина столбца.

В столбце `key_len` отражается максимально возможная длина индексированных полей, а не фактическая длина данных, хранимых в таблице. Так, в предыдущем примере MySQL всегда показывает 13 байт, даже если столбец `a` не содержит более одного символа. Иными словами, значение `key_len` вычисляется исходя из определения структуры таблицы, а не значений в ней.

Столбец ref

Этот столбец показывает, какие столбцы и константы из предыдущих таблиц используются для поиска в индексе, имя которого указано в столбце key. В следующем примере демонстрируется сочетание условий соединения и псевдонимов. Обратите внимание на столбец ref, в котором показано, что таблица film фигурирует в тексте запроса под псевдонимом f:

```
mysql> EXPLAIN
    -> SELECT STRAIGHT_JOIN f.film_id
    -> FROM sakila.film AS f
    ->    INNER JOIN sakila.film_actor AS fa
    ->       ON f.film_id=fa.film_id AND fa.actor_id = 1
    ->    INNER JOIN sakila.actor AS a USING(actor_id);
...+-------+...+--------------------+---------+-----------------------+...
...| table |...| key                | key_len | ref                   |...
...+-------+...+--------------------+---------+-----------------------+...
...| a     |...| PRIMARY            | 2       | const                 |...
...| f     |...| idx_fk_language_id | 1       | NULL                  |...
...| fa    |...| PRIMARY            | 4       | const,sakila.f.film_id |...
...+-------+...+--------------------+---------+-----------------------+...
```

Столбец rows

В этом столбце демонстрируется, сколько строк, по оценке MySQL, придется прочитать, чтобы найти запрошенные. Это значение вычисляется в расчете на каждую итерацию плана выполнения с вложенными циклами. Иными словами, это не просто количество строк, которые, по мнению MySQL, предстоит прочитать, а среднее количество строк, которые нужно будет прочитать, чтобы удовлетворить критерию, действующему в данной точке выполнения запроса. (Критерием может быть как константа, заданная в SQL-команде, так и текущее значение столбца из предыдущей таблицы в порядке соединения.)

Оценка может оказаться очень неточной, это зависит от имеющейся статистики таблицы и избирательности индекса. Кроме того, в версии MySQL 5.0 и более ранних не учитывается ключевое слово LIMIT. Например, при обработке следующего запроса не будут просматриваться 1022 строки:

```
mysql> EXPLAIN SELECT * FROM sakila.film LIMIT 1\G
 ...
        rows: 1022
```

Грубо оценить количество просматриваемых строк для всего запроса можно, перемножив все значения в столбце rows. Например, при выполнении следующего запроса, возможно, понадобится просмотреть примерно 2600 строк:

```
mysql> EXPLAIN
    -> SELECT f.film_id
    -> FROM sakila.film AS f
    ->    INNER JOIN sakila.film_actor AS fa USING(film_id)
```

```
    ->    INNER JOIN sakila.actor AS a USING(actor_id);
...+------+...
...| rows |...
...+------+...
...|  200 |...
...|   13 |...
...|    1 |...
...+------+...
```

Напомним, что речь идет не о числе строк в результирующем наборе, а о количестве строк, которое MySQL должна будет просмотреть. Кроме того, имейте в виду, что существует множество оптимизаций, например буферы соединения и кэши, которые не учитываются при оценивании количества строк. Очень может быть, что число реально читаемых сервером строк окажется меньше предсказанного. К тому же MySQL ничего не знает о кэшах, реализованных на уровне операционной системы и оборудования.

Столбец filtered

Этот столбец появился в версии MySQL 5.1 и отображается при выполнении команды `EXPLAIN EXTENDED`. Он показывает пессимистическую оценку процентной доли строк, удовлетворяющих некоторому критерию, заданному, например, в разделе `WHERE` или в условии соединения. Если умножить значение в столбце `rows` на этот процент, то получится оценка числа строк, которые MySQL должна будет соединить с предыдущими таблицами. В настоящее время оптимизатор использует эту оценку только для методов доступа `ALL`, `index`, `range` и `index_merge`.

Для иллюстрации мы создали следующую таблицу:

```
CREATE TABLE t1 (
    id INT NOT NULL AUTO_INCREMENT,
    filler char(200),
    PRIMARY KEY(id)
);
```

Затем вставили в нее 1000 строк со случайным текстом в столбце `filler`. Его назначение состоит в том, чтобы MySQL не использовала покрывающий индекс при выполнении следующего запроса:

```
mysql> EXPLAIN EXTENDED SELECT * FROM t1 WHERE id < 500\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1000
     filtered: 49.40
        Extra: Using where
```

MySQL могла бы воспользоваться методом доступа `range` для поиска всех строк, в которых поле `id` меньше 500, но не стала так поступать, потому что при этом удалось бы исключить только половину строк. Сервер полагает, что полное сканирование таблицы будет менее затратно. При этом лишние строки отфильтровываются по условию, указанному в разделе `WHERE`. Имея оценку стоимости, вычисленную для просмотра диапазона, сервер знает, сколько строк это условие отсеет. Поэтому-то в столбце `filtered` и отображается значение 49,40 %.

Столбец Extra

Этот столбец содержит дополнительную информацию, для которой не нашлось места в других столбцах. В руководстве по MySQL документированы многие значения, отображаемые в указанном столбце, часть из них упоминалась на страницах этой книги.

Наиболее важны следующие выражения, с которыми вам придется сталкиваться чаще всего.

- `Using index`. Означает, что MySQL воспользуется покрывающим индексом, чтобы избежать доступа к самой таблице. Не путайте покрывающие индексы с методом доступа по индексу.
- `Using where`. Означает, что сервер произведет дополнительную фильтрацию строк, отобранных подсистемой хранения. Многие условия `WHERE`, в которых имеются индексированные столбцы, могут быть проверены подсистемой хранения, когда (и если) она читает индекс, поэтому не для всех запросов со словом `WHERE` признак `Using where` существует. Иногда его наличие означает лишь, что запрос можно было бы выполнить более эффективно при другом индексировании.
- `Using temporary`. Означает, что MySQL будет применять временную таблицу для сортировки результатов запроса.
- `Using filesort`. Означает, что MySQL прибегнет к внешней сортировке для упорядочения результатов, а не станет читать строки из таблицы в порядке, задаваемом индексом. В MySQL реализованы два алгоритма файловой сортировки, о которых можно прочитать в главе 6. В обоих случаях сортировка может быть произведена в памяти или на диске. `EXPLAIN` ничего не говорит о том, какой тип файловой сортировки будет использован и где именно это произойдет.
- `range checked for each record (index map: N)`. Означает, что подходящего индекса не нашлось, поэтому сервер будет заново искать индекс при обработке каждой строки в операции соединения. *N* представляет собой битовую карту индексов, показанных в столбце `possible_keys`, так что эта информация избыточна.

Вывод плана выполнения в виде дерева

Пользователи MySQL говорили, что неплохо было бы выводить результат работы `EXPLAIN` в виде дерева, поскольку оно может более точно отразить структуру плана выполнения. Существующее представление не очень удобно для изучения плана, но древовидное отображение сложно совместить с табличным форматом. Неудоб-

ство особенно наглядно проявляется в большом количестве возможных значений в столбце `Extra`, а также в представлении `UNION`. Конструкция `UNION` совершенно не похожа на другие виды соединений, выполняемых MySQL, поэтому плохо сочетается с форматом `EXPLAIN`.

Хорошо разбираясь в различных правилах и особенностях `EXPLAIN`, можно восстановить изначально древовидный план выполнения. Но это утомительное занятие, которое лучше оставить автоматическому инструменту. Как раз такой инструмент pt-visual-explain имеется в пакете Percona Toolkit.

Улучшения в версии MySQL 5.6

MySQL 5.6 будет включать в себя важное улучшение команды `EXPLAIN` — возможность объяснять такие команды, как `UPDATE`, `INSERT` и т. п. Это очень полезно, поскольку, хотя команду DML можно преобразовать в квазиэквивалентный запрос с ключевым словом `SELECT` и объяснить его, результат не будет отражать то, как на самом деле выполняется исходная команда. При разработке и использовании инструментов, таких как pt-upgrade из пакета Percona Toolkit, которые пытаются использовать эту методику, мы столкнулись с несколькими случаями, когда оптимизатор не шел по пути, которого мы от него ожидали при преобразовании команд в `SELECT`. Таким образом, возможность использования `EXPLAIN` для команды без преобразования ее в `SELECT` помогает понять, что действительно происходит во время выполнения.

MySQL 5.6 также будет включать в себя множество улучшений оптимизатора запросов и подсистемы выполнения, которые позволят создавать анонимные временные таблицы как можно позднее, вместо того чтобы всегда создавать и заполнять их перед оптимизацией и выполнением относящихся к ним частей запроса. Это позволит MySQL сразу же объяснять запросы с подзапросами, не выполняя их предварительно.

Наконец, в MySQL 5.6 связанные с оптимизатором области будут улучшены путем добавления серверу функциональности *трассировки оптимизатора*. Это позволит пользователю просматривать принятые оптимизатором решения, а также входные параметры (например, кардинальность индекса) и причины принятия решений. Это будет очень полезно для понимания не только плана выполнения, выбранного сервером, но и причин такого выбора.

Приложение Д. Отладка блокировок

В любой системе, где для управления совместным доступом к разделяемым ресурсам применяются блокировки, очень трудно отлаживать возникающие вследствие конкуренции за их получение ошибки. Возможно, вы пытаетесь добавить в таблицу новый столбец или просто запускаете запрос и внезапно обнаруживаете, что операция не выполняется, потому что кто-то другой заблокировал таблицу или необходимые вам строки. Зачастую нужно лишь понять, почему запрос заблокирован, но иногда хочется определить, кто его заблокировал, чтобы знать, какой процесс убивать. В этом приложении мы покажем, как решить обе задачи.

Ожидание блокировки на уровне сервера

Блокировки могут происходить как на уровне сервера, так и на уровне подсистемы хранения[1]. (Блокировки на уровне приложения тоже могут представлять проблему, но сейчас мы говорим только о сервере MySQL.) Сам сервер MySQL применяет несколько типов блокировок.

- **Табличные блокировки.** Таблицы можно блокировать, устанавливая явные блокировки чтения и записи. Существует несколько разновидностей подобных блокировок, например локальные блокировки чтения. Прочитать обо всех разновидностях можно в разделе руководства по MySQL, посвященном команде `LOCK TABLES`. Помимо явных блокировок, запросы захватывают неявные на время своего выполнения.
- **Глобальные блокировки.** Существует единственная глобальная блокировка, захватываемая командой `FLUSH TABLES WITH READ LOCK` либо установкой параметра `read_only = 1`. Эта блокировка конфликтует с табличными.
- **Блокировки на имя.** Блокировка на имя — это разновидность табличной блокировки, которую сервер захватывает, когда переименовывает или удаляет таблицу.
- **Строковые блокировки.** Можно захватить и освободить блокировку на произвольную строку на уровне сервера, воспользовавшись функцией `GET_LOCK()` и родственными ей.

В последующих разделах мы рассмотрим все эти блокировки более подробно.

---

[1] Если вы забыли о том, как разделяются обязанности между сервером и подсистемами хранения, посмотрите на рис. 1.1.

Табличные блокировки

Табличные блокировки могут быть явными и неявными. Явная блокировка захватывается командой `LOCK TABLES`. Например, если в сеансе работы с программой mysql выполнить следующую команду:

```
mysql> LOCK TABLES sakila.film READ;
```

то будет захвачена явная блокировка на таблицу `sakila.film`.

Если затем в другом сеансе выполнить показанную далее команду:

```
mysql> LOCK TABLES sakila.film WRITE;
```

то запрос повиснет и не завершится.

На первом соединении вы увидите ожидающий поток:

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
      Id: 7
    User: baron
    Host: localhost
      db: NULL
 Command: Query
    Time: 0
   State: NULL
    Info: SHOW PROCESSLIST
*************************** 2. row ***************************
      Id: 11
    User: baron
    Host: localhost
      db: NULL
 Command: Query
    Time: 4
   State: Locked
    Info: LOCK TABLES sakila.film WRITE
2 rows in set (0.01 sec)
```

Обратите внимание на то, что поток 11 находится в состоянии `Locked`. В коде MySQL есть только одно место, где поток входит в это состояние: там, где он пытается получить блокировку на таблицу, которая в данный момент заблокирована другим потоком. Таким образом, увидев подобную картину, можно сразу же сказать, что поток ждет блокировки на уровне сервера, а не какой-то подсистемы хранения.

Однако явные блокировки не единственная причина остановки подобной операции. Как мы уже отмечали, сервер неявно блокирует таблицы на время выполнения запросов. Продемонстрировать это проще всего, запустив длительный запрос, для чего достаточно применить функцию `SLEEP()`:

```
mysql> SELECT SLEEP(30) FROM sakila.film LIMIT 1;
```

Если попытаться еще раз заблокировать таблицу `sakila.film`, пока работает этот запрос, то операция повиснет из-за неявной блокировки — точно так же, как это

произошло бы в случае явной. В списке процессов вы будете наблюдать такую же ситуацию, как и ранее:

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
      Id: 7
    User: baron
    Host: localhost
      db: NULL
 Command: Query
    Time: 12
   State: Sending data
    Info: SELECT SLEEP(30) FROM sakila.film LIMIT 1
*************************** 2. row ***************************
      Id: 11
    User: baron
    Host: localhost
      db: NULL
 Command: Query
    Time: 9
   State: Locked
    Info: LOCK TABLES sakila.film WRITE
```

В этом примере неявно захваченная при выполнении команды `SELECT` блокировка чтения препятствует получению явной блокировки на запись командой `LOCK TABLES`. Неявные блокировки могут блокировать и друг друга.

Может возникнуть вопрос: чем неявные блокировки отличаются от явных? На внутреннем уровне их структура одинакова, и управляются они одним и тем же кодом. А на внешнем вы можете управлять явными блокировками с помощью команд `LOCK TABLES` и `UNLOCK TABLES`.

Однако, когда речь заходит о подсистемах хранения, отличных от MyISAM, между этими блокировками обнаруживается одно очень важное различие. Блокировка, созданная явно, делает именно то, о чем вы попросили, тогда как неявные блокировки скрыты и ведут себя магическим образом. Сервер захватывает и освобождает неявные блокировки автоматически по мере необходимости и сообщает о них подсистеме хранения. Подсистема хранения конвертирует эти блокировки, исходя из своих потребностей. Например, в InnoDB существуют правила, диктующие, какую табличную блокировку следует создавать для данной табличной блокировки на уровне сервера. Поэтому трудно понять, какие же блокировки InnoDB создает в реальности.

Определение владельца блокировки. Если вы видите много процессов в состоянии `Locked`, то, возможно, дело в том, что MyISAM или похожая подсистема хранения применяется в условиях рабочей нагрузки с высокой конкурентностью. Это может помешать ручному выполнению той или иной операции, например построению нового индекса. Если в очереди на получение блокировки для таблицы типа MyISAM стоит запрос `UPDATE`, то не могут выполняться даже `SELECT`-запросы. (Дополнительную информацию об очередях на получение блокировки и о приоритетах можно почерпнуть из руководства по MySQL.)

Иногда становится очевидно, что какое-то соединение удерживает блокировку на таблицу слишком долго и его просто нужно принудительно разорвать (или воззвать

к совести пользователя и попросить, чтобы он не мешал работать всем остальным!). Но как определить, какое соединение владеет блокировкой?

В настоящее время не существует SQL-команды, показывающей, какой поток удерживает табличные блокировки, которые не дают выполнить ваш запрос. Команда `SHOW PROCESSLIST` демонстрирует только процессы, которые ждут блокировки, но не процессы, которые их захватили. К счастью, существует команда `debug`, которая выводит информацию о блокировках в журнал ошибок сервера. Чтобы запустить ее, воспользуйтесь утилитой mysqladmin:

```
$ mysqladmin debug
```

Она выводит довольно много отладочной информации, и ближе к концу находится то, что нас интересует. Эта распечатка получена после того, как мы заблокировали таблицу в одном соединении и затем попробовали заблокировать ее же в другом:

```
Thread database.table_name Locked/Waiting   Lock_type
7      sakila.film         Locked - read    Read lock  without concurrent inserts
8      sakila.film         Waiting - write  Highest priority write lock
```

Как видите, поток 8 ожидает блокировку, удерживаемую потоком 7.

Глобальная блокировка чтения

Сервер MySQL реализует также глобальную блокировку чтения. Получить ее можно следующим образом:

```
mysql> FLUSH TABLES WITH READ LOCK;
```

Если теперь попытаться заблокировать любую таблицу в другом сеансе, то этот сеанс повиснет:

```
mysql> LOCK TABLES sakila.film WRITE;
```

Как узнать, ждет запрос глобальную блокировку чтения или табличную блокировку? Следует взглянуть на результат команды `SHOW PROCESSLIST`:

```
mysql> SHOW PROCESSLIST\G
...
*************************** 2. row ***************************
      Id: 22
    User: baron
    Host: localhost
      db: NULL
 Command: Query
    Time: 9
   State: Waiting for release of readlock
    Info: LOCK TABLES sakila.film WRITE
```

Обратите внимание на состояние процесса `Waiting for release of readlock` (Ожидает освобождения блокировки чтения). Это как раз и означает, что запрос ждет глобальную блокировку чтения, а не табличную блокировку.

MySQL не позволяет узнать, кто удерживает глобальную блокировку чтения.

Блокировки на имя

Блокировка на имя — это разновидность табличной блокировки, которую сервер захватывает, когда собирается переименовать или удалить таблицу. Такая блокировка конфликтует с обычной табличной блокировкой, все равно, явной или неявной. Например, если мы, как и раньше, выполним в одном сеансе команду `LOCK TABLES`, а в другом попытаемся эту таблицу переименовать, то запрос повиснет, но на этот раз не в состоянии Locked:

```
mysql> RENAME TABLE sakila.film2 TO sakila.film;
```

Посмотрев на заблокированный запрос в списке процессов, мы увидим, что он находится в состоянии `Waiting for table` (Ожидает таблицу):

```
mysql> SHOW PROCESSLIST\G
...
*************************** 2. row ***************************
      Id: 27
    User: baron
    Host: localhost
      db: NULL
 Command: Query
    Time: 3
   State: Waiting for table
    Info: rename table sakila.film to sakila.film 2
```

Результат блокировки на имя можно увидеть также, применив команду `SHOW OPEN TABLES`:

```
mysql> SHOW OPEN TABLES;
+----------+-----------+--------+-------------+
| Database | Table     | In_use | Name_locked |
+----------+-----------+--------+-------------+
| sakila   | film_text |      3 |           0 |
| sakila   | film      |      2 |           1 |
| sakila   | film2     |      1 |           1 |
+----------+-----------+--------+-------------+
3 rows in set (0.00 sec)
```

Обратите внимание на то, что заблокированы оба имени — и старое и новое. Имя `sakila.film_text` заблокировано, потому что к таблице `sakila.film` присоединен ссылающийся на нее триггер. Таким образом, мы видим, что блокировки могут проявляться там, где их и не ожидаешь. При обращении к таблице `sakila.film` триггер вызывает неявное обращение к `sakila.film_text`, а стало быть, и захват неявной блокировки на нее. Вообще-то при переименовании триггер не должен срабатывать, поэтому, строго говоря, блокировка не нужна, но так уж сложилось: уровень детализации блокировок в MySQL не всегда такой, каким хотелось бы его видеть.

MySQL не позволяет узнать, кто удерживает блокировку на имя, но обычно это несущественно, так как такие блокировки очень быстротечны. Конфликт может быть вызван скорее тем, что саму блокировку на имя не удается получить из-за того, что кто-то захватил обычную табличную блокировку, но это как раз можно узнать с помощью `mysqladmin debug`, как показано ранее.

Пользовательские блокировки

И последняя разновидность блокировок, реализованная на уровне сервера, — пользовательские блокировки, по существу являющиеся не чем иным, как именованными мьютексами. Вы задаете строку, на которую нужно получить блокировку, и время ожидания в секундах, по истечении которого произойдет тайм-аут:

```
mysql> SELECT GET_LOCK('my lock', 100);
+--------------------------+
| GET_LOCK('my lock', 100) |
+--------------------------+
|                        1 |
+--------------------------+
1 row in set (0.00 sec)
```

Эта попытка удалась с первого раза, так что теперь поток владеет данным именованным мьютексом. Если другой поток попытается получить блокировку на ту же самую строку, то он подвиснет до истечения тайм-аута. В списке процессов для него показывается специальное состояние:

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
      Id: 22
    User: baron
    Host: localhost
      db: NULL
 Command: Query
    Time: 9
   State: User lock
    Info: SELECT GET_LOCK('my lock', 100)
```

Состояние `User lock` характерно только для таких блокировок. MySQL не позволяет узнать, кто владеет пользовательской блокировкой.

Ожидание блокировки в InnoDB

Блокировки на уровне сервера отлаживать проще, чем блокировки внутри подсистемы хранения. Механизм блокирования зависит от самой подсистемы, а некоторые из них вообще не предоставляют средств для получения информации о своих блокировках. В этом приложении мы будем говорить в основном об InnoDB.

InnoDB раскрывает информацию о блокировках с помощью команды `SHOW INNODB STATUS`. Если транзакция ожидает некоторую блокировку, то последняя будет присутствовать в секции `TRANSACTIONS`. Например, выполнив следующие команды в одном сеансе, вы захватите блокировку записи на первую строку таблицы:

```
mysql> SET AUTOCOMMIT=0;
mysql> BEGIN;
mysql> SELECT film_id FROM sakila.film LIMIT 1 FOR UPDATE;
```

Если теперь выполнить те же команды в другом сеансе, то запрос повиснет в ожидании блокировки, захваченной в первом. `SHOW INNODB STATUS` продемонстрирует, что произошло (для простоты мы приводим только часть распечатки):

```
1  LOCK WAIT 2 lock struct(s), heap size 1216
2  MySQL thread id 8, query id 89 localhost baron Sending data
3  SELECT film_id FROM sakila.film LIMIT 1 FOR UPDATE
4  ------- TRX HAS BEEN WAITING 9 SEC FOR THIS LOCK TO BE GRANTED:
5  RECORD LOCKS space id 0 page no 194 n bits 1072 index `idx_fk_language_id`
   of table `sakila/film` trx id 0 61714 lock_mode X waiting
```

В последней строке показано, что запрос ожидает монопольную (`lock_mode X`) блокировку на страницу 194 индекса `idx_fk_language_id` данной таблицы. В конечном итоге тайм-аут ожидания истечет и запрос вернет такую ошибку:

```
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

К сожалению, не зная, кто удерживает блокировку, трудно понять, какая транзакция стала источником проблемы. Иногда можно сделать обоснованное предположение, посмотрев, какие транзакции открыты очень давно. Можно также активировать монитор блокировок InnoDB, который показывает до десяти блокировок, удерживаемых каждой транзакцией. Чтобы это сделать, нужно создать таблицу типа InnoDB со специальным именем[1]:

```
mysql> CREATE TABLE innodb_lock_monitor(a int) ENGINE=INNODB;
```

После выполнения этого запроса InnoDB начинает периодически (с разной частотой, но обычно несколько раз в минуту) помещать в стандартный поток вывода ту же информацию, что и команда `SHOW INNODB STATUS`, но в слегка расширенном формате. В большинстве систем стандартный поток вывода перенаправлен в журнал ошибок сервера, поэтому, изучив его, вы сможете узнать, какие транзакции удерживают блокировки. Чтобы остановить монитор, удалите эту таблицу.

Вот пример выдаваемой информации:

```
1  ---TRANSACTION 0 61717, ACTIVE 3 sec, process no 5102, OS thread id 1141152080
2  3 lock struct(s), heap size 1216
3  MySQL thread id 11, query id 108 localhost baron
4  show innodb status
5  TABLE LOCK table `sakila/film` trx id 0 61717 lock mode IX
6  RECORD LOCKS space id 0 page no 194 n bits 1072 index `idx_fk_language_id` of
   table `sakila/film` trx id 0 61717 lock_mode X
7  Record lock, heap no 2 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
8   ... опущено ...
9
10  RECORD LOCKS space id 0 page no 231 n bits 168 index `PRIMARY` of table
    `sakila/film` trx id 0 61717 lock_mode X locks rec but not gap
11  Record lock, heap no 2 PHYSICAL RECORD: n_fields 15; compact format; info bits 0
12   ... опущено...
```

[1] InnoDB понимает несколько волшебных имен таблиц. Сейчас предпочитают динамически изменяемые серверные переменные, но InnoDB существует уже давно, поэтому сохранила некоторые старые привычки.

Обратите внимание на строку 3, где показан идентификатор потока MySQL, — то же самое значение, что в столбце `Id` в списке процессов. В строке 5 видно, что транзакция владеет неявной монопольной блокировкой (`IX`) на таблицу. В строках 6–8 демонстрируется блокировка на индекс. Информацию в строке 8 мы опустили, потому что это довольно длинный дамп заблокированной записи. В строках 9–11 показана блокировка на первичный ключ (фраза `FOR UPDATE` приводит к блокировке не только индекса, но и самой строки).

При активизированном мониторе блокировок дополнительная информация появляется и в распечатке, формируемой командой `SHOW INNODB STATUS`, поэтому не обязательно заглядывать в журнал ошибок сервера.

Монитор блокировок не самое оптимальное решение по нескольким причинам. Основная проблема — выдача чрезмерно подробной информации, которая включает дамп заблокированных записей в шестнадцатеричном и ASCII-виде. Все это быстро заполняет журнал ошибок и легко может переполнить буфер вывода `SHOW INNODB STATUS`, размер которого фиксирован. А тогда вы не увидите именно то, что ищете, поскольку эти сведения находятся в последних секциях. Кроме того, в InnoDB зашито ограничение на количество выводимых блокировок для одной транзакции (десять), поэтому сведений об интересующей вас блокировке вы можете как раз и не увидеть. Но даже если искомое имеется, найти его отнюдь не всегда просто (попробуйте активизировать монитор на загруженном сервере и убедитесь сами!).

Сделать формат вывода информации о блокировках более удобным можно двумя путями. Во-первых, один из авторов этой книги написал заплату для InnoDB и сервера MySQL, которая вошла в Percona Server и MariaDB. Эта заплата удаляет из выводимой информации подробные дампы записей, включает информацию о блокировках в команду `SHOW INNODB STATUS` по умолчанию (поэтому монитор вообще не нужно активизировать) и добавляет динамически изменяемые серверные переменные для управления уровнем подробности и количеством блокировок, печатаемых для одной транзакции.

Второй способ — воспользоваться программой innotop для разбора и форматирования вывода. В режиме Lock она показывает блокировки, сгруппированные по соединению и по таблице, так что можно без труда понять, какие транзакции удерживают блокировку на данную таблицу. Это небезупречный метод поиска виновной транзакции, так как для обнаружения заблокированной записи необходимо исследовать дампы записей. Но все же он гораздо лучше имеющихся альтернатив и достаточно хорош в большинстве случаев.

Использование таблиц INFORMATION_SCHEMA. Использование `SHOW INNODB STATUS` для просмотра блокировок — это, безусловно, устаревшая практика, поскольку теперь у InnoDB есть таблицы `INFORMATION_SCHEMA`, которые раскрывают свои транзакции и блокировки.

Если вы не видите таблицы, значит, используете старую версию InnoDB. Вам нужны хотя бы MySQL 5.1 и плагин InnoDB. Если вы используете MySQL 5.1 и все равно не видите таблицу `INNODB_LOCKS`, проверьте `SHOW VARIABLES` для переменной

innodb_version. Если не видите переменную, значит, не используете плагин InnoDB, а это необходимо! Если вы видите переменную, но нет таблиц, вам необходимо убедиться, что параметр plugin_load в файле конфигурации сервера явно включает таблицы. Чтобы узнать об этом подробнее, обратитесь к руководству по MySQL.

К счастью, в MySQL 5.5 вам не нужно беспокоиться обо всем этом — современная версия InnoDB встроена в сервер.

В руководствах MySQL и InnoDB есть примеры запросов, с которыми нужно обращаться к этим таблицам. Здесь мы не будем их повторять, однако приведем парочку своих собственных. Например, вот запрос, который показывает, кто блокирует, кто ждет и как долго:

```
SELECT r.trx_id AS waiting_trx_id,  r.trx_mysql_thread_id AS waiting_thread,
       TIMESTAMPDIFF(SECOND, r.trx_wait_started, CURRENT_TIMESTAMP) AS wait_time,
       r.trx_query AS waiting_query,
       l.lock_table AS waiting_table_lock,
       b.trx_id AS blocking_trx_id, b.trx_mysql_thread_id AS blocking_thread,
       SUBSTRING(p.host, 1, INSTR(p.host, ':') - 1) AS blocking_host,
       SUBSTRING(p.host, INSTR(p.host, ':') +1) AS blocking_port,
       IF(p.command = "Sleep", p.time, 0) AS idle_in_trx,
       b.trx_query AS blocking_query
FROM INFORMATION_SCHEMA.INNODB_LOCK_WAITS  AS w
INNER JOIN INFORMATION_SCHEMA.INNODB_TRX   AS b ON  b.trx_id = w.blocking_trx_id
INNER JOIN INFORMATION_SCHEMA.INNODB_TRX   AS r ON  r.trx_id = w.requesting_trx_id
INNER JOIN INFORMATION_SCHEMA.INNODB_LOCKS AS l ON  w.requested_lock_id = l.lock_id
LEFT JOIN  INFORMATION_SCHEMA.PROCESSLIST  AS p ON  p.id     = b.trx_mysql_thread_id
ORDER BY wait_time DESC\G
*************************** 1. row ***************************
    waiting_trx_id: 5D03
    waiting_thread: 3
         wait_time: 6
     waiting_query: select * from store limit 1 for update
waiting_table_lock: `sakila`.`store`
   blocking_trx_id: 5D02
   blocking_thread: 2
     blocking_host: localhost
     blocking_port: 40298
       idle_in_trx: 8
    blocking_query: NULL
```

Результат показывает, что поток 3 ждет 6 секунд для блокировки строки в таблице store. Она была заблокирована потоком 2, который простаивает в течение 8 секунд.

Если у вас большие проблемы из-за большого числа блокировок, возникших из-за простаивающих в транзакциях потоков, узнать, правда без детализации, сколько запросов заблокировано и на каких потоках, поможет следующий вариант:

```
SELECT CONCAT('thread ', b.trx_mysql_thread_id, ' from ', p.host) AS who_blocks,
       IF(p.command = "Sleep", p.time, 0) AS idle_in_trx,
       MAX(TIMESTAMPDIFF(SECOND, r.trx_wait_started, NOW())) AS max_wait_time,
       COUNT(*) AS num_waiters
FROM INFORMATION_SCHEMA.INNODB_LOCK_WAITS  AS w
INNER JOIN INFORMATION_SCHEMA.INNODB_TRX   AS b ON  b.trx_id = w.blocking_trx_id
INNER JOIN INFORMATION_SCHEMA.INNODB_TRX   AS r ON  r.trx_id = w.requesting_trx_id
```

```
LEFT JOIN  INFORMATION_SCHEMA.PROCESSLIST  AS p ON  p.id     = b.trx_mysql_thread_id
GROUP BY who_blocks ORDER BY num_waiters DESC\G
*************************** 1. row ***************************
    who_blocks: thread 2 from localhost:40298
   idle_in_trx: 1016
 max_wait_time: 37
   num_waiters: 8
```

Результат показывает, что поток 2 теперь простаивает намного дольше и по крайней мере один поток ждет до 37 секунд, пока он освободит свои блокировки. Имеется восемь потоков, ожидающих, пока поток 2 завершит свою работу и совершит коммит.

Мы обнаружили, что блокировка простоя в транзакции является общей причиной чрезвычайных проблем и часто ее трудно обнаружить. Инструмент pt-kill из пакета Percona Toolkit можно настроить так, чтобы он прерывал длительные бездействующие транзакции для предотвращения такой ситуации. Кроме того, для достижения той же цели в Percona Server имеется параметр ожидания тайм-аута транзакции.

Приложение E. Использование Sphinx совместно с MySQL

Sphinx (http://www.sphinxsearch.com) — это бесплатная полнотекстовая поисковая система с открытым исходным кодом, которая изначально спроектирована в расчете на интеграцию с базами данных. Она содержит ряд средств, характерных для СУБД, работает очень быстро, поддерживает распределенный поиск и хорошо масштабируется. Кроме того, в проект заложены эффективное использование памяти и минимизация дискового ввода/вывода, что особенно важно, так как часто именно эти факторы ограничивают производительность крупных операций.

Sphinx прекрасно работает в сочетании с MySQL. Ее можно использовать для ускорения разнообразных запросов, в том числе полнотекстовых, а также для выполнения быстрой сортировки и группировки — и это далеко не все возможности ее применения. Она использует сетевой протокол MySQL и в основном совместимый с MySQL диалект SQL, поэтому, работая с ней, можно реализовывать такие же запросы, как и в ходе взаимодействия с базой данных MySQL. Кроме того, существует реализованная в виде подключаемого модуля подсистема хранения, с помощью которой программист или администратор может обращаться к Sphinx напрямую из MySQL. Система Sphinx особенно полезна для некоторых запросов, которые MySQL не очень хорошо оптимизирует для больших наборов данных вследствие универсальности своей архитектуры. Короче говоря, Sphinx дополняет функциональность сервера MySQL и повышает его производительность.

Источником данных для индекса Sphinx обычно является запрос SELECT к MySQL, но, вообще говоря, можно построить индекс на основе неограниченного количества источников разных типов, а каждый экземпляр Sphinx способен искать в произвольном числе индексов. Например, можно включить в индекс некоторые документы из базы данных MySQL, работающей на одном удаленном сервере, документы из базы данных PostgreSQL, работающей на другом сервере, и данные, поставляемые локальным сценарием по XML-конвейеру.

В этом приложении рассмотрим ряд случаев, когда возможности Sphinx позволяют повысить производительность, опишем порядок установки и конфигурирования, подробно расскажем о функциях системы и обсудим реальные примеры практического применения.

Типичный поиск с помощью Sphinx

Начнем с простого, но довольно показательного примера использования Sphinx, чтобы наметить отправную точку для дальнейшего обсуждения. Мы воспользуемся языком PHP из-за его популярности, но API существует и для ряда других языков.

Предположим, что нужно реализовать полнотекстовый поиск для использования в системе сравнения товаров, к которой предъявляются следующие требования:

- поддерживать полнотекстовый индекс для поиска по таблице товаров, хранящейся в базе данных MySQL;
- обеспечить полнотекстовый поиск по названию и описанию каждого товара;
- при необходимости выполнять поиск только в пределах заданной категории;
- иметь возможность сортировать результаты не только по релевантности, но и по цене товара или дате поступления в продажу.

Для начала опишем источник данных и индекс в конфигурационном файле Sphinx:

```
source products
 {
     type         = mysql
     sql_host     = localhost
     sql_user     = shopping
     sql_pass     = mysecretpassword
     sql_db       = shopping
     sql_query    = SELECT id, title, description, \
                    cat_id, price, UNIX_TIMESTAMP(added_date) AS added_ts \
                    FROM products
     sql_attr_uint        = cat_id
     sql_attr_float       = price
     sql_attr_timestamp   = added_ts
 }

index products
 {   source    = products
     path      = /usr/local/sphinx/var/data/products
     docinfo   = extern
}
```

В этом примере предполагается, что в базе данных `shopping` имеется таблица `products`, из которой мы выбираем в запросе `SELECT` столбцы для заполнения индекса Sphinx. Сам индекс Sphinx также называется `products`. Создав новый источник данных и индекс, мы запускаем программу indexer, которая создает начальный полнотекстовый индекс, а затем запускаем или перезапускаем программу-демон searchd, которая подхватит изменения:

```
$ cd /usr/local/sphinx/bin
$ ./indexer products
$ ./searchd --stop
$ ./searchd
```

Теперь индекс готов отвечать на запросы. Чтобы протестировать его, запустим скрипт `test.php`, входящий в дистрибутив Sphinx:

```
$ php -q test.php -i products ipod

Query 'ipod ' retrieved 3 of 3 matches in 0.010 sec.
Query stats:
     'ipod' found 3 times in 3 documents
```

```
Matches:
1. doc_id=123, weight=100, cat_id=100, price=159.99, added_ts=2008-01-03 22:38:26
2. doc_id=124, weight=100, cat_id=100, price=199.99, added_ts=2008-01-03 22:38:26
3. doc_id=125, weight=100, cat_id=100, price=249.99, added_ts=2008-01-03 22:38:26
```

Последний шаг — добавить функцию поиска в веб-приложение. Мы должны настроить параметры сортировки и фильтрации в соответствии с пожеланиями пользователя и красиво отформатировать результаты. Кроме того, поскольку Sphinx возвращает клиенту только идентификаторы документов и прописанные в конфигурационном файле атрибуты (он не хранит исходные текстовые данные), то недостающую информацию мы должны получить от MySQL самостоятельно:

```
<?php
include ( "sphinxapi.php" );
// ... прочие директивы include, код подключения к MySQL,
// вывод заголовка страницы, формы поиска и т. п. — все это здесь

// устанавливаем те параметры запроса, которые задал пользователь
$cl = new SphinxClient ();
$sortby = $_REQUEST["sortby"];
if ( !in_array ( $sortby, array ( "price", "added_ts" ) ) )
    $sortby = "price";
 if ( $_REQUEST["sortorder"]=="asc" )
    $cl->SetSortMode ( SPH_SORT_ATTR_ASC, $sortby );
 else
    $cl->SetSortMode ( SPH_SORT_ATTR_DESC, $sortby );
 $offset = ($_REQUEST["page"]-1)*$rows_per_page;
 $cl->SetLimits ( $offset, $rows_per_page );

 // отправляем запрос, получаем результаты
 $res = $cl->Query ( $_REQUEST["query"], "products" );

 // обрабатываем ошибки поиска
 if ( !$res )
 {
    print "<b>Search error:</b>" . $cl->GetLastError ();
    die;
 }

 // выбираем дополнительные столбцы из базы данных MySQL
 $ids = join ( ",", array_keys ( $res["matches"] );
 $r = mysql_query ( "SELECT id, title FROM products WHERE id IN ($ids)" )
    or die ( "MySQL error: " . mysql_error() );
 while ( $row = mysql_fetch_assoc($r) )
 {
    $id = $row["id"];
    $res["matches"][$id]["sql"] = $row;
 }

 // выводим результаты в том порядке, в котором их вернула система Sphinx
 $n = 1 + $offset;
 foreach ( $res["matches"] as $id=>$match )
 {
    printf ( "%d. <a href=details.php?id=%d>%s</a>, USD %.2f<br>\n",
```

```
43          $n++, $id, $match["sql"]["title"], $match["attrs"]["price"] );
44  }
45
46  ?>
```

Хотя показанный код несложен, но кое-что в нем все же следует пояснить.

- Функция `SetLimits()` говорит Sphinx, что необходимо извлечь лишь то количество результатов, которое клиент желает вывести на странице. В Sphinx такое ограничение обходится дешево (в отличие от поиска, встроенного в MySQL), а число результатов, которое было бы возвращено, не будь этого ограничения, можно получить, не нагружая систему дополнительно, из переменной `$result['total_found']`.
- Поскольку Sphinx только индексирует столбец `title`, но не хранит его, мы должны извлекать данные непосредственно из базы MySQL.
- Мы выбираем значения из базы MySQL одним запросом для всех документов, перечисляя их идентификаторы в разделе `WHERE id IN (...)`, а не выполняем отдельные запросы для каждого найденного документа (это было бы неэффективно).
- Полученные от MySQL строки мы вставляем в результат полнотекстового поиска, чтобы сохранить исходный порядок сортировки. Чуть позже мы еще вернемся к этому пункту.
- Отображаемые строки сконструированы из значений, полученных как от Sphinx, так и от MySQL.

Код включения данных в отображаемые строки специфичен для PHP и заслуживает чуть более подробного объяснения. Мы не можем просто обойти результирующий набор, возвращенный сервером MySQL, поскольку порядок строк в нем может (и, как правило, так и происходит) отличаться от последовательности идентификаторов в разделе `WHERE id IN (...)`. Однако в хеше (ассоциативном массиве) PHP результаты хранятся в том порядке, в котором были туда помещены, поэтому при обходе `$result["matches"]` мы формируем строки в той последовательности, в какой их вернула система Sphinx. Таким образом, чтобы сохранить нужный порядок результатов (вместо полуслучайного порядка, в котором возвращает строки MySQL), мы вставляем полученные от MySQL данные по одному в хеш, где PHP хранит результирующий набор Sphinx.

Между MySQL и Sphinx имеется существенная разница как в реализации, так и в производительности подсчета количества найденных результатов и применения ключевого слова `LIMIT`. Во-первых, сразу отметим, что `LIMIT` обходится в Sphinx очень дешево. Рассмотрим `LIMIT 500,10`. MySQL выбрала бы 510 первых попавшихся строк (это медленно) и 500 из них отбросила, тогда как Sphinx возвращает идентификаторы, с помощью которых вы позже выберете из базы MySQL именно те десять строк, которые нужны. Во-вторых, Sphinx всегда возвращает точное количество найденных записей независимо от того, что задано ключевым словом `LIMIT`. MySQL не умеет делать это эффективно, хотя в версии 5.6 произведены некоторые улучшения.

Зачем использовать Sphinx

Sphinx может дополнить приложение на основе MySQL разными способами, демонстрируя высокую производительность там, где MySQL не блещет, и предлагая функциональность, которой MySQL не располагает. Перечислим некоторые типичные сценарии использования.

- Быстрый, эффективный, масштабируемый и релевантный полнотекстовый поиск.
- Оптимизация условий `WHERE`, когда индекс отсутствует или обладает низкой селективностью.
- Оптимизация запросов с фразами `ORDER BY ... LIMIT N` и `GROUP BY`.
- Параллельная генерация результирующих наборов.
- Вертикальное и горизонтальное масштабирование.
- Агрегирование секционированных данных.

В следующих разделах мы рассмотрим все эти сценарии. Но этот список неисчерпывающий, и пользователи постоянно находят Sphinx новое применение. Например, один из наиболее важных способов практической работы с системой Sphinx — быстрый поиск и фильтрация записей — результат творчества пользователей, он не закладывался при проектировании.

Эффективный и масштабируемый полнотекстовый поиск

Средства полнотекстового поиска, встроенные в MySQL, хороши для небольших наборов данных, но с ростом базы начинают работать очень медленно. Когда количество записей исчисляется миллионами, а объем индексированного текста — гигабайтами, время выполнения запроса может варьироваться от секунды до 10 минут и более, что для высокопроизводительного веб-приложения, конечно, неприемлемо. Полнотекстовый поиск средствами MySQL допускает масштабирование за счет распределения данных по нескольким серверам, но организовывать параллельный поиск и объединение найденных результатов придется в приложении.

Sphinx работает значительно быстрее, чем встроенные в MySQL полнотекстовые индексы. Например, на поиск в тексте размером свыше 1 Гбайт затрачивается 10–100 миллисекунд, а при объеме 10–100 Гбайт время линейно зависит от количества процессоров. Кроме того, система Sphinx обладает следующими достоинствами.

- Может индексировать данные, хранящиеся в InnoDB и других подсистемах хранения, а не только в MyISAM.
- Может создавать индексы по данным, взятым из многих таблиц, а не только из столбцов одной таблицы.
- Может динамически объединять результаты поиска по нескольким индексам.
- Помимо текстовых столбцов, индексы могут содержать неограниченное количество числовых атрибутов, их можно считать в некотором роде дополнительными

столбцами. В качестве значений атрибутов допускается использовать целые числа, числа с плавающей точкой и временные метки UNIX.

- Умеет оптимизировать полнотекстовый поиск с дополнительными условиями на атрибуты.
- Алгоритм ранжирования, учитывающий полные фразы, позволяет возвращать наиболее релевантные результаты. Например, при поиске в таблице песен о любви по запросу «Я люблю тебя, родная» песня, содержащая в точности эту фразу, окажется в начале списка — раньше тех, которые несколько раз содержат слова «люблю» и «родная».
- Значительно упрощает горизонтальное масштабирование.

Эффективное применение условия WHERE

Иногда требуется сделать выборку из очень больших таблиц (с миллионами записей), а в запросе присутствуют условия WHERE для столбцов, по которым построены индексы с очень низкой селективностью (то есть для заданного условия возвращается слишком много строк) или не построены вообще. Типичные случаи — поиск пользователей в социальной сети или лотов на аукционном сайте. Как правило, в форме поиска можно задать десять и более столбцов, а сортироваться результаты могут совсем по другим столбцам. Пример такого приложения и соответствующей стратегии индексирования см. в главе 5.

Если правильно спроектировать схему и провести оптимизацию запросов, то MySQL вполне прилично справляется с такими ситуациями, если в условии WHERE не слишком много столбцов. Но по мере роста количества столбцов число индексов, необходимых для поддержки всех возможных комбинаций, растет экспоненциально. Полное покрытие всех возможных сочетаний всего четырех столбцов находится в опасной близости к пределу возможностей MySQL. К тому же поддержка таких индексов — очень медленный и затратный процесс. Следовательно, практически бессмысленно строить все требуемые индексы для поддержки условий WHERE с большим числом столбцов.

Но еще важнее тот факт, что, даже добавив индексы, толку от них вы получите немного, если они обладают низкой селективностью. Классический пример — столбец gender, который мало чем помогает, потому что каждому значению соответствует примерно половина всех строк. Если селективность индекса недостаточна, то MySQL обычно предпочитает полное сканирование таблицы.

Sphinx выполняет такие запросы гораздо быстрее, чем MySQL. При построении индекса Sphinx можно указать только требуемые столбцы из таблицы с данными. Sphinx допускает два способа доступа к таблицам: поиск ключевого слова по индексу или полное сканирование. В обоих случаях применяются *фильтры* — эквивалент условия WHERE. Но в отличие от MySQL, которая сама определяет, использовать индекс или прибегнуть к полному сканированию, Sphinx позволяет пользователю выбирать метод доступа.

Чтобы воспользоваться полным сканированием с фильтрами, следует задать в качестве критерия поиска пустую строку. Если же нужен поиск по индексу, то при его построении включите в поисковые поля псевдоключевые слова, а затем ищите по ним. Например, если требуется найти товары в категории 123, то на этапе индексирования нужно добавить в документ ключевое слово `категория123` и затем искать по нему. Добавлять слова можно в одно из уже имеющихся полей с помощью функции `CONCAT()`, а можно для большей гибкости создать специальное поисковое поле для псевдоключевых слов. Обычно стоит использовать фильтры, когда критерий неселективен, то есть ему удовлетворяет больше 30 % строк, и фиктивные ключевые слова — когда критерию удовлетворяет менее 10 % строк. В серой зоне от 10 до 30 % возможно всякое, так что для поиска оптимального решения следует воспользоваться эталонным тестированием.

Sphinx выполняет и поиск по индексу, и полное сканирование быстрее, чем MySQL. Иногда для полного сканирования Sphinx требуется даже меньше времени, чем MySQL на чтение из индекса.

Поиск первых нескольких результатов с учетом сортировки

В веб-приложениях часто нужно искать первые *N* результатов с учетом заданной сортировки. Как мы писали ранее, этот случай плохо поддается оптимизации в MySQL 5.5 и более старых версиях.

Хуже всего дело обстоит, когда поиск по указанному условию `WHERE` возвращает много строк (допустим, 1 миллион), а столбцы, перечисленные в разделе `ORDER BY`, не проиндексированы. MySQL воспользуется индексом для поиска всех подходящих строк, будет читать их одну за другой в буфер сортировки, собирая по всему диску, затем произведет файловую сортировку — и все это только для того, чтобы бóльшую часть отбросить. На время обработки запроса MySQL вынуждена сохранять и обрабатывать все строки, игнорируя ограничение `LIMIT` и потребляя огромное количество памяти. А если результат не помещается в буфер сортировки, то придется прибегнуть к сортировке на диске, что лишь увеличивает число операций дискового ввода/вывода.

Это крайний случай, и может даже показаться, что в реальной жизни такое случается редко, но проблемы, которые он иллюстрирует, возникают сплошь и рядом. MySQL ограничивает множество индексов, пригодных для сортировки: допускается использование только левой части ключа, не поддерживается неплотный просмотр индекса и разрешается только одно условие с указанием диапазона. Поэтому многие реальные запросы ничего не выигрывают от наличия индексов. А даже если выигрывают, то дисковый ввод/вывод для извлечения строк в полуслучайном порядке способен обнулить производительность.

Еще одной проблемой в MySQL является разбиение результирующего набора на страницы, для которого обычно требуется выполнять запросы типа `SELECT ... LIMIT N, M`. При этом с диска читается *N* + *M* строк и, следовательно, производится много опера-

ций произвольной выборки, вследствие чего впустую расходуется память. Sphinx может существенно ускорить выполнение таких запросов, устраняя две самые крупные проблемы.

- **Потребление памяти.** Потребление памяти в Sphinx всегда строго ограничено, причем лимиты можно конфигурировать. Sphinx поддерживает размер и смещение результирующего набора аналогично конструкции `LIMIT N, M`, применяемой в MySQL, но допускает также параметр `max_matches`. Он управляет размером аналога буфера сортировки как на уровне сервера, так и на уровне отдельного запроса. Гарантируется, что потребление памяти в Sphinx не превышает заданных ограничений.
- **Ввод/вывод.** Если атрибуты находятся в памяти, то Sphinx вообще не обращается к диску. Но даже если атрибуты хранятся на диске, то для их считывания Sphinx выполняет последовательный ввод/вывод, что гораздо быстрее извлечения в полуслучайном порядке, характерного для MySQL.

Результаты поиска можно сортировать по комбинации релевантности (веса), значений атрибутов и (при использовании `GROUP BY`) значений агрегатной функции. Синтаксис задания порядка сортировки аналогичен разделу `ORDER BY` в SQL:

```
<?php
$cl = new SphinxClient ();
$cl->SetSortMode ( SPH_SORT_EXTENDED, 'price DESC, @weight ASC' );
// дополнительный код и вызов Query() ...
?>
```

В этом примере `price` — это заданный пользователем атрибут, хранящийся в индексе, а `@weight` — специальный атрибут, создаваемый во время выполнения и содержащий вычисленную релевантность результата. Можно сортировать также по арифметическому выражению, включающему значения атрибутов, стандартные математические операции и функции:

```
<?php
$cl = new SphinxClient ();
$cl->SetSortMode ( SPH_SORT_EXPR, '@weight + log(pageviews)*1.5' );
// дополнительный код и вызов Query() ...
?>
```

Оптимизация запросов, содержащих GROUP BY

Поддержка типичных для языка SQL фраз была бы неполной без функциональности `GROUP BY`, поэтому в Sphinx она тоже реализована. Но в отличие от универсального механизма, применяемого в MySQL, Sphinx специализируется на эффективном решении полезного на практике подмножества задач с помощью конструкции `GROUP BY`. Это подмножество покрывает генерацию отчетов по большим (1–100 миллионов строк) наборам данных в случаях, когда выполняется одно из следующих условий.

- Результирующий набор составляет лишь малую толику от общего числа группируемых строк (здесь под малой толикой может пониматься от 100 тысяч до 1 миллиона строк).

- Требуется очень высокая скорость работы, поэтому можно смириться с тем, что значение `COUNT(*)` будет приближенным, если много групп извлекается из данных, распределенных между машинами, входящими в кластер.

Эти условия не настолько строго ограничивают, как может показаться. Первый случай покрывает практически все мыслимые отчеты по времени. Например, для построения детального почасового отчета за десять лет нужно извлечь менее 90 000 записей. Второй случай можно на естественном языке описать так: «Найти 20 наиболее значимых записей в шардированой таблице из 100 миллионов строк настолько быстро и точно, насколько это возможно».

Эта техника позволяет ускорить выполнение запросов общего вида, но ее можно использовать и в приложениях с полнотекстовым поиском. Часто бывает нужно не просто показать результаты полнотекстового поиска, но и агрегировать их тем или иным способом. Например, на многих страницах поисковой выдачи отображается количество результатов, найденных в каждой категории товаров, или выводится график изменения счетчика обнаруженных документов в зависимости от времени. Еще одно типичное требование — сгруппировать результаты и показать наиболее релевантные в каждой категории. Поддержка `GROUP BY` в Sphinx позволяет комбинировать группировку с полнотекстовым поиском и тем самым избежать издержек на группировку в приложении или в MySQL.

Как и для сортировки, для группировки в Sphinx отведена память фиксированного размера. Она работает немного эффективнее (на 10–50 %), чем аналогичные запросы в MySQL, в случае, когда набор данных целиком умещается в памяти. При таком условии преимущество системы Sphinx обусловлено ее способностью распределять нагрузку и существенно уменьшать время задержки. Для очень больших наборов данных, которые в память никак не уместить, можно в целях формирования отчетов построить специальный индекс на диске с помощью встраиваемых атрибутов (подробнее об этом позже). Запросы к таким индексам выполняются примерно с той же скоростью, с какой можно считывать данные с диска: на современном оборудовании — 30–100 Мбит/с. В таком случае производительность оказывается многократно выше, чем у MySQL, хотя результаты получаются приближенными.

Самое важное различие между реализациями `GROUP BY` в MySQL и Sphinx состоит в том, что Sphinx при определенных условиях дает примерные результаты. Тому есть две причины.

- Для группировки используется память фиксированного размера. Если групп больше, чем может поместиться в памяти, а результаты возвращаются в некотором «несчастливом» порядке, то счетчики по группам могут оказаться меньше истинных значений.
- При распределенном поиске между узлами пересылаются только агрегированные результаты, а не сами найденные документы. Если в разных узлах имеются записи-дубликаты, то счетчики различающихся записей в каждой группе могут оказаться больше истинных значений, поскольку информация, которая позволила бы устранить дубликаты, между узлами не передается.

На практике часто можно смириться с приближенными счетчиками по группам, лишь бы они быстро вычислялись. Если же это недопустимо, то часто удается получить точные результаты, соответствующим образом настроив демон и клиентское приложение.

Можно также получить эквивалент конструкции `COUNT(DISTINCT <атрибуты>)`. Например, этим можно воспользоваться на аукционном сайте для подсчета различных продавцов в каждой категории.

Наконец, Sphinx позволяет задать критерий для выбора одного наилучшего документа в каждой группе. В частности, в ходе группировки по домену и сортировки результирующего набора по счетчику найденных в домене можно выбрать наиболее релевантный документ в каждом домене. В MySQL для этого понадобился бы очень сложный запрос.

Генерация параллельных результирующих наборов

Sphinx позволяет одновременно генерировать несколько результирующих наборов из одних и тех же данных, опять-таки не выходя за пределы ограничений по памяти. Это дает ощутимый выигрыш по сравнению с традиционно применяемым в SQL подходом, когда либо запускаются два запроса (в надежде, что после первого какие-то данные останутся в кэше), либо для каждого результирующего набора создается временная таблица.

Например, предположим, что нужно сгенерировать отчеты по дням, неделям и месяцам за некоторый период. Для этого в MySQL пришлось бы выполнить три запроса с разными разделами `GROUP BY`, то есть обратиться к источнику данных трижды. Sphinx умеет генерировать все три отчета параллельно, обрабатывая источник всего один раз. Для этого в Sphinx предусмотрен механизм *мультизапросов*. Вместо того чтобы выдавать запросы по одному, вы собираете их в пакет и отправляете один раз:

```
<?php
$cl = new SphinxClient ();
$cl->SetSortMode ( SPH_SORT_EXTENDED, "price desc" );
$cl->AddQuery ( "ipod" );
$cl->SetGroupBy ( "category_id", SPH_GROUPBY_ATTR, "@count desc" );
$cl->AddQuery ( "ipod" );
$cl->RunQueries ();
?>
```

Sphinx проанализирует поступивший запрос, выделит из него отдельные части и, если возможно, распараллелит их выполнение.

Например, Sphinx может заметить, что различаются только режимы сортировки и группировки, а в остальном запросы одинаковы. Именно так обстоит дело в примере, приведенном ранее, где сортировка производится по полю `price`, а группировка — по `category_id`. Для обработки таких запросов Sphinx создаст несколько очередей сортировки. Затем каждая выбранная строка будет помещена во все очереди.

По сравнению с последовательным выполнением запросов это устраняет излишние операции полнотекстового поиска или полного сканирования.

Отметим, что генерация параллельных результирующих наборов, будучи распространенной и важной оптимизацией, является лишь частным случаем более общего механизма мультизапросов. Это не единственная возможная оптимизация. Можно посоветовать всюду, где возможно, объединять несколько запросов в один пакет, так как это позволяет Sphinx применить различные внутренние оптимизации. Даже если Sphinx не сумеет распараллелить выполнение запросов, все равно количество обращений к серверу уменьшится. А если в будущих версиях появятся новые оптимизации, то они будут применены к вашим запросам автоматически, так что вам не потребуется ничего изменять.

Масштабирование

Sphinx отлично масштабируется как горизонтально, так и вертикально.

Sphinx можно распределять по нескольким компьютерам без ограничений. Во всех вышеупомянутых способах ее применения можно получить выигрыш от распределения нагрузки между несколькими процессорами.

Поисковый демон Sphinx (searchd) поддерживает специальные распределенные индексы, которые знают, какие локальные и удаленные индексы следует опрашивать и агрегировать. Это означает, что горизонтальное масштабирование становится тривиальной конфигурационной задачей. Нужно лишь распределить данные по узлам и сконфигурировать главный, который будет рассылать удаленные запросы, выполняющиеся параллельно с локальными, и все.

Допустимо также вертикальное масштабирование, когда с целью уменьшения задержки на одной машине устанавливается большее количество процессоров или ядер. Для этого нужно лишь запустить на одном компьютере несколько экземпляров демона searchd и опрашивать их все с другого компьютера через распределенный индекс. Или сконфигурировать единственный экземпляр так, чтобы он общался сам с собой, тогда параллельные «удаленные» запросы на самом деле будут выполняться на одной машине, но разными процессорами или ядрами.

Иными словами, в Sphinx можно сделать так, что один запрос будет исполняться несколькими процессорами (несколько параллельных запросов автоматически обрабатываются разными процессорами). Это основное отличие от MySQL, где одному запросу всегда выделяется один процессор вне зависимости от того, сколько их имеется в наличии. Кроме того, Sphinx не нуждается в синхронизации между параллельно выполняющимися запросами. Это позволяет отказаться от мьютексов (механизм синхронизации), которые являются печально известной причиной многих узких мест в работе MySQL на машине с несколькими процессорами.

Еще один важный аспект вертикального масштабирования — это масштабирование дискового ввода/вывода. Чтобы повысить пропускную способность и уменьшить

задержку, различные индексы, в том числе части большого распределенного индекса, можно без проблем поместить на разные физические диски или в разные тома RAID. Этот подход обладает некоторыми достоинствами механизма секционированных таблиц в версии MySQL 5.1, который также позволяет размещать данные в нескольких местах. Однако у распределенных индексов есть преимущества по сравнению с секционированными таблицами. Sphinx использует их как для разделения нагрузки, так и для параллельной обработки частей запроса. Что же касается механизма секционирования в MySQL, то он позволяет оптимизировать некоторые (но не все) запросы за счет отсечения секций, но никак не распараллелить обработку. И хотя методики секционирования и в Sphinx, и в MySQL повышают пропускную способность, но для запросов, требующих большого объема ввода/вывода, от Sphinx можно ожидать линейного снижения задержки всегда, а от MySQL — только в тех случаях, когда оптимизатору удается отсечь некоторые секции целиком.

Последовательность операций при распределенном поиске проста.

1. Разослать запросы всем удаленным серверам.
2. Выполнить последовательный локальный поиск по индексу.
3. Получить частичные результаты поиска от удаленных серверов.
4. Объединить все частичные результаты в окончательный и вернуть его клиенту.

Если оборудование позволяет, то можно запустить параллельный поиск по набору индексов и на одной машине. При наличии нескольких физических дисков и процессорных ядер эти одновременно выполняющиеся поиски не будут мешать друг другу. Можно сделать вид, что некоторые индексы являются удаленными, и сконфигурировать searchd так, чтобы он обращался сам к себе для запуска параллельного запроса на той же машине:

```
index distributed_sample
{
     type = distributed
     local = chunk1 # resides on HDD1
     agent = localhost:3312:chunk2 # resides on HDD2, searchd contacts itself
}
```

С точки зрения клиента распределенные индексы абсолютно ничем не отличаются от локальных. Это позволяет создавать деревья распределенных индексов, используя одни узлы как прокси для других. Например, узел первого уровня мог бы транслировать запросы нескольким узлам второго уровня, которые, в свою очередь, могли бы либо обрабатывать их локально либо передавать запрос дальше. Причем глубина дерева не ограничена.

Агрегирование шардированных данных

При построении масштабируемой системы часто применяется шардирование (секционирование) информации на несколько физических серверов MySQL. Этот вопрос мы подробно обсуждали в главе 11.

Если данные шардированы детально, то просто для выборки нескольких строк с селективным условием WHERE (вообще-то это быстрая операция) необходимо обратиться к нескольким серверам, проконтролировать ошибки и объединить результаты в самом приложении. Sphinx отчасти решает эту проблему, так как вся необходимая функциональность уже реализована внутри поискового демона.

Рассмотрим пример. Пусть таблица размером 1 Тбайт, содержащая 1 миллиард постов в блогах, шардирована по идентификатору пользователя и размещена на десяти физических серверах MySQL так, что все сообщения одного пользователя всегда оказываются на одном и том же сервере. Если в запросе фигурирует только один пользователь, то все хорошо: выбираем сервер по идентификатору пользователя и дальше работаем как обычно.

Но предположим, что требуется реализовать страницы архива, на которых отображаются сообщения друзей данного пользователя. Как мы будем выводить другие функции утилиты sysbench для записей с 981-й по 1000-ю, отсортированные по дате сообщения? Скорее всего, данные, относящиеся к разным друзьям, находятся на разных серверах. Если друзей всего десять, то вероятность того, что понадобится обращаться более чем к восьми серверам, составляет 90 %, а когда друзей 20, она возрастает до 99 %. Таким образом, для большинства запросов придется опрашивать все серверы. Более того, нам предстоит отобрать 1000 сообщений с каждого сервера и отсортировать их в приложении. Следуя рекомендациям, приведенным ранее в этой книге, мы могли бы свести необходимые сведения только к идентификатору сообщения и временной метке, но все равно придется отсортировать 10 000 записей внутри приложения. У большинства современных скриптовых языков только на эту сортировку уйдет масса процессорного времени. И не забудем, что нужно будет опрашивать серверы последовательно (это медленно) или написать какой-то код, который запустит параллельные потоки (а его трудно реализовать и сопровождать).

В таких ситуациях имеет смысл не изобретать велосипед, а обратиться к Sphinx. Всего-то и нужно запустить несколько экземпляров Sphinx, скопировать наиболее востребованные атрибуты из каждой таблицы — в данном случае идентификатор сообщения, его дату и идентификатор пользователя — и запросить у главного экземпляра Sphinx записи с 981-й по 100-ю, отсортированные по дате публикации. На все про все требуется три строчки кода. Такой способ масштабирования куда разумнее.

Обзор архитектуры

Система Sphinx представляет собой автономный набор программ. Из них наиболее важны две:

- indexer. Это приложение извлекает документы из указанных источников (например, из результата выполнения запроса к MySQL) и строит по ним полнотекстовый индекс. Обычно оно периодически запускается в пакетном режиме;
- searchd. Программа-демон, которая выполняет запросы к индексам, построенным программой indexer. Приложения обращаются к ней во время выполнения.

В дистрибутив Sphinx входят также API естественных клиентов searchd для нескольких языков программирования (на момент написания книги в этот список входили PHP, Python, Perl, Ruby и Java) и клиент SphinxSE, реализованный в виде подключаемой подсистемы хранения для версии MySQL 5.0 и более поздних. Данные API и SphinxSE позволяют клиентскому приложению подключиться к демону searchd, передать ему поисковый запрос и получить результаты поиска.

Полнотекстовый индекс Sphinx можно сравнить с таблицей базы данных, но состоит он не из строк, а из документов. (В Sphinx есть также отдельная структура данных, называемая многозначным атрибутом, мы обсудим ее позже.) У каждого документа есть уникальный 32- или 64-разрядный целочисленный идентификатор, который должен извлекаться из индексируемой таблицы базы данных (например, из столбца, содержащего первичный ключ). Кроме того, в каждом документе имеются одно или несколько полнотекстовых полей, каждое из которых соответствует одному текстовому столбцу в базе данных, и числовых атрибутов. Как и в таблице базы данных, в одном индексе Sphinx состав полей и атрибутов во всех документах одинаков. В табл. E.1 проведена аналогия между таблицей базы данных и индексом Sphinx.

Таблица E.1. Соответствие между структурами базы данных и индекса Sphinx

Структура базы данных	Структура индекса Sphinx
CREATE TABLE documents (id int(11) NOT NULL auto_increment, title varchar(255), content text, group_id int(11), added datetime, PRIMARY KEY (id));	index documents document ID title field, full-text indexed content field, full-text indexed group_id attribute, sql_attr_uint added attribute, sql_attr_timestamp

Sphinx не хранит текстовые поля из базы данных, а лишь использует их содержимое для построения индекса.

Общие сведения об установке

Установка Sphinx не вызывает затруднений и обычно сводится к следующим шагам.

1. Сборка программ из исходных текстов:

```
$ configure && make && make install
```

2. Создание конфигурационного файла, содержащего описания источников данных и полнотекстовые индексы.
3. Первоначальное индексирование.
4. Запуск searchd.

После этого функциональность поиска сразу же становится доступна клиентским программам:

```
<?php
include ( 'sphinxapi.php' );
$cl = new SphinxClient ();
$res = $cl->Query ( 'test query', 'myindex' );
// здесь можно использовать результат поиска $res
?>
```

Остается только регулярно запускать indexer для обновления полнотекстового индекса. Индексы, с которыми в данный момент работает демон searchd, остаются полностью функциональными во время переиндексации: indexer видит, что они используются, создает теневой индекс и по завершении уведомляет searchd, что нужно переключиться на него.

Полнотекстовые индексы хранятся в файловой системе (в каталоге, который указан в конфигурационном файле) в специальном монолитном формате, плохо приспособленном для инкрементных обновлений. Обычный способ обновления данных в индексе — полная перестройка. Но это не такая серьезная проблема, как могло бы показаться, по следующим причинам.

- Индексирование производится быстро. На современном оборудовании система Sphinx способна индексировать простой текст (без HTML-разметки) со скоростью 4–8 Мбит/с.
- В следующем разделе показано, что данные можно секционировать на несколько индексов и при каждом запуске indexer заново индексировать только изменившиеся части.
- Нет необходимости дефрагментировать индексы — они строятся в расчете на оптимальное использование подсистемы ввода/вывода, что повышает скорость поиска.
- Числовые атрибуты можно обновлять без полной перестройки индекса.

В будущих версиях планируется реализовать дополнительные средства, которые будут поддерживать обновление индексов в реальном масштабе времени.

Типичное применение секционирования

Давайте рассмотрим секционирование более подробно. Простейшая схема секционирования называется *«главный + разностный»*, при этом по одному и тому же набору документов создаются два индекса. Главный индекс построен по всем документам, а разностный — только по тем, которые изменились с момента построения главного индекса.

Эта схема прекрасно подходит для многих вариантов модификации данных. В качестве примеров можно привести форумы, блоги, архивы почты и новостей, вертикальные поисковые системы. Данные в этих репозиториях по большей части не изменяются после ввода, а число регулярно добавляемых или модифицируемых документов мизерно по сравнению с общим количеством. Поэтому разностный индекс очень мал,

и его можно перестраивать часто (например, раз в 1–15 минут). Это эквивалентно индексированию только вновь добавленных строк.

Для изменения ассоциированных с документами атрибутов нет необходимости перестраивать индексы — это можно сделать оперативно с помощью searchd. Чтобы пометить строку как удаленную, достаточно просто установить атрибут `deleted` в главном индексе. А для обработки обновлений нужно будет пометить этим атрибутом документы в главном индексе, а затем перестроить разностный. Тогда поиск всех документов, не имеющих признака `deleted`, вернет правильный результат.

Отметим, что данные для индексирования могут поступать в виде результата выполнения произвольной команды `SELECT`, и не обязательно только из одной таблицы. На структуру команды `SELECT` не накладывается никаких ограничений. Это означает, что перед индексированием можно обработать данные в базе. Обычно такая предварительная обработка включает соединение с другими таблицами, создание дополнительных полей на лету, исключение некоторых полей из индекса и манипулирование значениями.

Специальные возможности

Помимо простого индексирования и поиска по содержимому базы данных, Sphinx предлагает ряд специальных возможностей. Перечислим наиболее важные.

- Алгоритмы поиска и ранжирования принимают во внимание позиции слов и близость поисковой фразы к содержимому документа.
- К документам можно привязывать числовые атрибуты, в том числе многозначные.
- Разрешается сортировать, фильтровать и группировать по значениям атрибутов.
- Можно создавать фрагменты документов, в которых поисковые слова подсвечены.
- Поиск можно производить сразу на нескольких машинах.
- Можно оптимизировать запросы, порождающие сразу несколько результирующих наборов из одних и тех же данных.
- Подсистема хранения SphinxSE позволяет обращаться к результатам поиска непосредственно из MySQL.
- Можно настраивать нагрузку на сервер со стороны Sphinx.

Часть этих средств мы рассматривали ранее, а в данном разделе обсудим некоторые из оставшихся.

Ранжирование по близости фразы

Как и другие поисковые системы с открытым исходным кодом, Sphinx запоминает позиции начала каждого слова в документе. Но в отличие от большинства других она использует эти позиции также для ранжирования результатов поиска, что позволяет возвращать более релевантные результаты.

На окончательный ранг документа может повлиять множество факторов. В большинстве систем для ранжирования используется только частота появления поисковых слов, то есть учитывается, сколько раз каждое слово встречалось в документе. Классическая весовая функция BM25[1], которую применяют практически все полнотекстовые поисковые системы, придает больший вес тем словам, которые чаще встречаются в конкретном тексте или редко встречаются во всем наборе документов. Обычно результат, возвращенный функцией BM25, считается окончательным значением ранга.

Но Sphinx вычисляет также близость поисковой фразы, то есть величину самого длинного отрезка фразы, встречающегося в документе, выраженную в количестве слов. Например, близость фразы John Doe Jr к документу, содержащему текст John Black, John White Jr, and Jane Dunne, равна 1, потому что никакие два слова, указанные в запросе, не встречаются в нем подряд и в том же порядке, что и в ключевой фразе. Тот же запрос в применении к документу, содержащему предложение Mr. John Doe Jr and friends, дает близость 3, поскольку здесь присутствуют все три слова из запроса, причем в том же порядке. Близость искомой фразы к документу John Gray, Jane Doe Jr равна 2 благодаря наличию отрезка Doe Jr.

По умолчанию Sphinx ранжирует результаты сначала по близости фразы, а потом по классической весовой функции BM25. Следовательно, буквальные цитаты гарантированно окажутся первыми в списке, цитаты, отличающиеся только одним словом, — под ними и т. д.

Когда и как близость фразы влияет на результаты? Рассмотрим поиск выражения To be or not to be в 1 миллионе страниц текста. Sphinx поместит страницы, содержащие буквальную цитату, в начало списка результатов, тогда как системы на основе функции BM25 в первую очередь вернут страницы, где чаще всего встречаются слова to, be, or и not, документы же с точной цитатой, в которых слово to встречается всего несколько раз, будут зарыты в глубине результатов поиска.

Большинство современных крупных поисковых машин также ранжируют результаты поиска с учетом позиций ключевых слов. Поиск фразы в Google, скорее всего, поместит документы, содержащие буквальную и близкую цитаты, в начало списка, а уже потом будут располагаться тексты с «мешком слов».

Однако для анализа позиций ключевых слов требуется дополнительное время, и иногда этот этап желательно пропустить из соображений производительности. Бывает также, что ранжирование с учетом близости фразы дает неожиданные результаты. Например, поиск тегов в облаке лучше вести без учета позиций ключевых слов: не имеет значения, насколько близко друг к другу находятся теги, встретившиеся в документе.

Чтобы обеспечить необходимую гибкость, Sphinx предлагает несколько режимов ранжирования. Помимо принимаемого по умолчанию — близость плюс BM25, — можно выбрать и другие, в частности только весовая функция BM25, вообще без вычисления веса (неплохая оптимизация, если вы не собираетесь сортировать по рангу) и т. д.

[1] Подробнее см.: http://en.wikipedia.org/wiki/Okapi_BM25.

Поддержка атрибутов

Любой документ может содержать неограниченное количество числовых атрибутов. Атрибуты определяются пользователем и могут содержать дополнительную информацию, необходимую для решения конкретной задачи, например идентификатор автора сообщения в блоге, цену товара, идентификатор категории и т. д.

Атрибуты позволяют производить полнотекстовый поиск с дополнительной фильтрацией, сортировкой и группировкой результатов. Теоретически их можно было бы хранить в базе MySQL и извлекать оттуда каждый раз после завершения поиска. Но на практике, если полнотекстовый поиск находит хотя бы сотни или тысячи строк (не так уж много), последующая выборка их из базы будет недопустимо медленной.

Sphinx поддерживает два способа хранения атрибутов: встроенными в документ и в отдельном внешнем файле. В случае встраивания в индексе приходится хранить множество экземпляров значений атрибутов — по одному для каждого вхождения идентификатора документа. Это приводит к разбуханию индекса и увеличению объема операций ввода/вывода, но сокращает потребление памяти. Внешние же атрибуты требуется предварительно загружать в память на этапе запуска searchd.

Как правило, атрибуты помещаются в ОЗУ, так что обычно их хранят отдельно. При этом фильтрация, сортировка и группировка производятся очень быстро, поскольку для доступа к данным достаточно поиска в памяти. К тому же лишь внешние атрибуты можно обновлять во время выполнения. Встраивание атрибутов следует применять лишь тогда, когда для их хранения не хватает памяти.

Sphinx поддерживает также многозначные атрибуты (multivalued attributes (MVA)). Такой атрибут состоит из списка целочисленных значений, ассоциированных с каждым документом, имеющего произвольную длину. Примерами оправданного применения MVA могут служить списки идентификаторов тегов, категорий товаров и списки управления доступом.

Фильтрация

Имея доступ к атрибутам, Sphinx может использовать их для фильтрации и отвергать потенциальных кандидатов на ранних стадиях поиска. Технически фильтрация производится после того, как проверено, что документ содержит все заданные слова, но перед началом трудоемких вычислений, например ранжирования. Благодаря такой оптимизации применение Sphinx для комбинирования полнотекстового поиска с фильтрацией и сортировкой может оказаться в 10–100 раз быстрее, чем использование Sphinx только для поиска с последующей фильтрацией результатов средствами MySQL.

Sphinx поддерживает два типа фильтров, аналогичных простым условиям `WHERE` в SQL.

- Значение атрибута попадает в заданный диапазон (аналог ключевого слова `BETWEEN` или числового сравнения).
- Атрибут совпадает с одним из заданных значений (аналог списка `IN()`).

Если в фильтрах будет задаваться фиксированное количество величин (фильтры по набору, а не по диапазону) и эти значения будут селективными, целесообразно заменить целые числа фиктивными ключевыми словами и индексировать их как текст, а не как атрибуты. Это относится как к обычным, так и к многозначным числовым атрибутам. Примеры того, как это делается, приведем в дальнейшем.

Sphinx может использовать фильтры также для оптимизации полного сканирования. Sphinx запоминает минимальное и максимальное значения атрибута для короткого непрерывного блока строк (по умолчанию 128 строк) и может быстро отбрасывать целые блоки, не удовлетворяющие условиям фильтрации. Строки хранятся в порядке возрастания идентификаторов документов, поэтому такая оптимизация лучше всего работает, если значения столбцов коррелируют с идентификатором. Например, если в таблице имеется временная метка, которая возрастает вместе с идентификатором, то полное сканирование с фильтрацией по этой метке будет выполняться очень быстро.

Подключаемая подсистема хранения SphinxSE

Результаты полнотекстового поиска, полученные от Sphinx, почти всегда требуется затем обработать средствами MySQL — по крайней мере для того, чтобы извлечь значения текстовых столбцов, которые в Sphinx не хранятся. Поэтому нередко возникает необходимость соединить результаты поиска с другими таблицами MySQL, то есть выполнить операцию `JOIN`.

Это можно сделать, отправив MySQL запрос с идентификаторами найденных документов, однако такая стратегия не позволяет получить элегантный и быстрый код. Если речь идет о больших объемах данных, то лучше воспользоваться подсистемой хранения SphinxSE, которую можно включить при компиляции MySQL версии 5.0 или старше либо загрузить как подключаемый модуль для версии 5.1 и новее.

SphinxSE позволяет программисту опрашивать демон searchd и получать результаты поиска прямо из MySQL. Для этого достаточно создать специальную таблицу с указанием `ENGINE=SPHINX` (и необязательным параметром `CONNECTION`, который позволяет найти сервер Sphinx, если его нет там, где он подразумевается по умолчанию), а затем использовать ее в запросах:

```
mysql> CREATE TABLE search_table (
    ->     id        INTEGER NOT NULL,
    ->     weight    INTEGER NOT NULL,
    ->     query     VARCHAR(3072) NOT NULL,
    ->     group_id  INTEGER,
    ->     INDEX(query)
    -> ) ENGINE=SPHINX CONNECTION="sphinx://localhost:3312/test";
Query OK, 0 rows affected (0.12 sec)

mysql> SELECT * FROM search_table WHERE query='test;mode=all' \G
*************************** 1. row ***************************
       id: 123
   weight: 1
    query: test;mode=all
 group_id: 45
1 row in set (0.00 sec)
```

Каждая команда SELECT передает Sphinx запрос, текст которого находится в условии на столбец query в разделе WHERE (см. пример далее). Сервер searchd возвращает результаты. Подсистема хранения SphinxSE преобразует их в формат MySQL и возвращает команде SELECT.

В запросах можно употреблять операторы JOIN для соединения с другими таблицами любого типа.

Подсистема хранения SphinxSE поддерживает большинство возможностей поиска, доступных через API. Чтобы задать, к примеру, фильтрацию и лимиты, следует включить дополнительные параметры в строку запроса:

```
mysql> SELECT * FROM search_table WHERE query='test;mode=all;
    -> filter=group_id,5,7,11;maxmatches=3000';
```

Статистику по запросу и по словам, которую предоставляет API, можно получить также с помощью команды SHOW STATUS:

```
mysql> SHOW ENGINE SPHINX STATUS \G
*************************** 1. row ***************************
  Type: SPH INX
  Name: stats
Status: total: 3, total found: 3, time: 8, words: 1

*************************** 2. row ***************************
  Type: SPHINX
  Name: words Status: test:3:5
2 rows in set (0.00 sec)
```

Даже при использовании SphinxSE эмпирическое правило предлагает оставить сортировку, фильтрацию и группировку на откуп демону searchd, то есть включить все необходимые для этого параметры в строку запроса, а не в разделы WHERE, ORDER BY и GROUP BY. Особенно важно это для условий WHERE. Причина в том, что SphinxSE — всего лишь клиент searchd, а не полноценная встроенная библиотека поиска. Следовательно, для достижения максимальной производительности лучше передавать все что можно подсистеме Sphinx.

Средства управления производительностью

Индексирование и поиск могут создавать дополнительную нагрузку как на поисковый сервер, так и на сервер баз данных. Однако для ограничения нагрузки, создаваемой системой Sphinx, имеется целый ряд параметров.

Нежелательная нагрузка на базу данных может быть вызвана запросами от indexer, которые либо приводят к полной остановке работы MySQL из-за блокировок, либо поступают слишком быстро и отнимают ресурсы у конкурирующих запросов.

Первая ситуация характерна для MyISAM, где длительные блокировки чтения блокируют таблицы и приостанавливают другие операции чтения и записи — вы не сможете выполнить команду SELECT * FROM big_table на промышленном сервере, так как есть риск помешать выполнению другой работы. Чтобы обойти эту сложность, Sphinx поддерживает запросы с указанием диапазона. Вместо того чтобы конфигурировать

один гигантский запрос, можно задать запрос, который быстро вычислит диапазон значений в индексируемых строках, и другой запрос, который будет выбирать данные небольшими порциями:

```
sql_query_range  = SELECT MIN(id),MAX(id) FROM documents
sql_range_step   = 1000
sql_query        = SELECT id, title, body FROM documents \
                   WHERE id>=$start AND id<=$end
```

Эта возможность особенно полезна для индексирования таблиц типа MyISAM, но стоит подумать и о применении ее к таблицам InnoDB. Хотя при выполнении команды `SELECT *` к большому объему данных InnoDB не блокирует таблицы и не мешает работе других запросов, все равно из-за особенностей архитектуры MVCC потребляется много ресурсов. Многоверсионность для 1000 транзакций, каждая из которых охватывает 1000 строк, может потребовать меньших затрат, чем для одной длительной транзакции, затрагивающей 1 миллион строк.

Второй источник повышенной нагрузки проявляется, когда программа indexer обрабатывает данные быстрее, чем MySQL их поставляет. В этом случае также следует использовать запросы с указанием диапазона. Параметр `sql_ranged_throttle` заставляет indexer приостанавливать работу на заданный промежуток времени (в миллисекундах) перед отправкой очередного запроса. Время индексирования при этом увеличивается, зато нагрузка на MySQL снижается.

Интересно, что существует особый случай, когда имеет смысл настроить Sphinx для достижения прямо противоположного эффекта: сократить время индексирования за счет повышения нагрузки на MySQL. Если скорость сетевого соединения между машинами, на которых работают indexer и MySQL, составляет 100 Мбит/с и строки хорошо сжимаются (это типично для текстовых данных), то применение протокола сжатия MySQL может уменьшить общее время индексирования. Правда, за это как на стороне MySQL, так и на стороне indexer приходится расплачиваться дополнительным расходованием процессорного времени на упаковку и распаковку строк, передаваемых по сети. Тем не менее благодаря сокращению сетевого трафика время индексирования может уменьшиться на 20–30 %.

Поисковые кластеры также могут испытывать кратковременные перегрузки, поэтому Sphinx предоставляет несколько способов контроля нагрузки на searchd.

Во-первых, параметр `max_children` ограничивает количество одновременно выполняемых запросов. Когда заданный порог достигнут, клиенту предлагается повторить попытку позже.

Во-вторых, существуют лимиты на уровне запроса. Так, можно указать, что обработка запроса должна прекращаться, как только найдено максимальное количество документов или истекло отведенное время, для чего предназначены функции `SetLimits()` и `SetMaxQueryTime()` соответственно. Поскольку это делается на уровне запроса, то наиболее важные запросы можно выполнять до конца.

И наконец, при периодическом запуске indexer могут возникать всплески активности ввода/вывода, что замедляет работу searchd. Избежать этого позволяют параметры

для ограничения обращений indexer к диску. Параметр `max_iops` создает задержку между операциями ввода/вывода, ограничивая максимальное количество операций ввода/вывода в секунду своим значением. Но даже одной операции может оказаться слишком много (что, если она читает 100 Мбайт?). На этот случай есть параметр `max_iosize`, гарантирующий, что длина каждой операции чтения или записи не будет превышать заданного значения. Более длинные процессы автоматически разбиваются на части, к которым затем применяется параметр `max_iops`.

Примеры практической реализации

Все описанные выше средства нашли успешное практическое применение в той или иной системе. В следующих подразделах мы расскажем о нескольких реальных примерах использования Sphinx, упомянув о конкретных сайтах и о некоторых деталях реализации.

Полнотекстовый поиск на сайте Mininova.org

Популярная система поиска торрентов Mininova (**http://www.mininova.org**) показывает хороший пример оптимизации одного лишь полнотекстового поиска. Sphinx заменила встроенные полнотекстовые индексы на нескольких подчиненных серверах MySQL, которые не справлялись с объемами работы. После замены поисковые серверы оказались недогруженными, средняя нагрузка составляла 0,3–0,4.

Приведем сведения о размере базы данных и особенностях работы сервера.

- На этом сайте база данных невелика — 300 000–500 000 записей, а размер индекса составляет 300–500 Мбайт.
- Нагрузка на сайт довольно высока — 8–10 миллионов операций поиска в день (на момент написания книги).

Данные по большей части представляют собой заданные пользователями имена файлов, часто без соблюдения пунктуации. Поэтому вместо поиска по целым словам применяется поиск по префиксу. В результате индекс оказывается в несколько раз больше, чем мог бы быть, но все равно он довольно мал, так что строится быстро и допускает эффективное кэширование.

Результаты 1000 наиболее частых запросов кэшируются на уровне приложения. Примерно 20–30 % запросов обслуживаются из кэша. Поскольку для их распределения характерен длинный хвост, то увеличение размера кэша не помогло бы.

С целью повышения доступности сайт обслуживают два сервера, на каждом из которых хранятся идентичные реплики полнотекстовых индексов. Индексы перестраиваются полностью каждые несколько минут. Поскольку индексирование занимает меньше минуты, смысла в реализации более сложной схемы нет.

Из этого примера можно извлечь следующие уроки.

- Кэширование результатов поиска на уровне приложения значительно повышает производительность.
- Иногда нет необходимости в огромном кэше, даже если приложение сильно нагружено. Может хватать всего 1000–10 000 записей.
- Если размер базы данных составляет примерно 1 Гбайт, то даже для нагруженного сайта достаточно периодического полного переиндексирования вместо разработки более сложных схем.

Полнотекстовый поиск на сайте BoardReader.com

Mininova представляет одну крайность высоконагруженных проектов — данных мало, но к ним делается очень много запросов. Сайт BoardReader (http://www.boardreader.com) представляет собой другую крайность — это система поиска по форумам, которая выполняет гораздо меньше операций поиска, но на значительно более объемном наборе данных. Sphinx заменила коммерческую поисковую систему, которой требовалось до 10 секунд на поиск в наборе размером 1 Гбайт, и дала возможность масштабировать BoardReader в широких пределах — как по размеру, так и по пропускной способности.

Приведем информацию общего характера.

- Документов больше 1 миллиарда, объем текста в базе данных — порядка 1,5 Тбайт.
- Ежедневно просматривается примерно 500 000 страниц и выполняется от 700 000 до 1 миллиона операций поиска.

Когда создавалась эта книга, поисковый кластер состоял из шести серверов, каждый из которых был оснащен четырьмя логическими ЦП (два двухъядерных процессора Xeon), 16 Гбайт оперативной памяти и дисками совокупной емкостью 0,5 Тбайт. Сама база данных находилась на отдельном кластере. Поисковый кластер использовался только для индексирования и поиска.

На каждом из шести серверов запущено по четыре экземпляра searchd, чтобы были задействованы все четыре ядра. Один из четырех экземпляров агрегирует результаты, полученные остальными тремя. Всего действуют 24 экземпляра searchd. Данные между ними распределены равномерно. Каждая копия searchd обслуживает несколько индексов, в сумме составляющих примерно 1/24 общего объема данных (около 60 Гбайт).

Результаты поиска, возвращенные шестью узлами первого уровня, в свою очередь, агрегируются еще одним демоном searchd, который работает на веб-сервере. Этот экземпляр не содержит локальных данных, но обслуживает несколько распределенных индексов, которые ссылаются на шесть поисковых серверов, входящих в кластер.

Зачем нужны четыре экземпляра searchd на одном узле? Почему бы не запустить на каждом сервере по одному экземпляру, сконфигурировав его для обслуживания четырех групп индексов, и не заставить его посылать самому себе «удаленные» за-

просы, чтобы задействовать несколько ЦП, как мы советовали ранее? Но у четырех экземпляров вместо одного есть свои плюсы. Во-первых, сокращается время запуска. Существует несколько гигабайт атрибутивных данных, которые необходимо предварительно загрузить в память, и одновременный запуск нескольких демонов позволяет распараллелить эту задачу. Во-вторых, улучшается доступность. Когда выходит из строя или обновляется один экземпляр searchd, недоступной становится только 1/24 индекса, а не 1/6.

На каждом из 24 экземпляров searchd в поисковом кластере мы применяем секционирование по времени, чтобы еще больше сократить нагрузку. Многие запросы необходимо выполнять только для недавних данных, поэтому мы разделили весь объем информации на три непересекающихся набора индексов: данные за последнюю неделю, за последние три месяца и за все время. Эти индексы распределены по нескольким физическим дискам, отнесенным к отдельным экземплярам. Таким образом, у каждого экземпляра имеется собственный процессор и собственный физический диск, так что друг другу они не мешают.

Индексы периодически обновляются локальными заданиями cron. Данные извлекаются из MySQL по сети, но файлы индексов создаются локально.

Оказалось, что несколько отдельных неформатированных дисков работают быстрее одного тома RAID. В этом случае можно точно контролировать, на какой неформатированный диск записывается конкретный файл. В случае RAID это не так — здесь контроллер решает, какой блок на какой диск отправить. Кроме того, неформатированные диски гарантируют, что ввод/вывод в разные группы индексов будет полностью распараллелен, тогда как при одновременном запуске нескольких поисков на RAID-массиве возможна сериализация операций. Мы выбрали уровень RAID 0, не обеспечивающий резервирования, поскольку отказы дисков нас не заботили, так как индексы на поисковых серверах легко перестроить. Можно было бы использовать несколько томов RAID 1 (с зеркалированием) и получить ту же пропускную способность, какую дают отдельные диски, но с повышенной надежностью.

На примере BoardReader интересно отметить еще одну вещь — то, как выполняется переход на новую версию Sphinx. Очевидно, что останавливать кластер целиком нельзя. Поэтому критичной оказывается обратная совместимость. К счастью, Sphinx ее обеспечивает — новые версии searchd могут читать старые индексы и взаимодействовать по сети со старыми клиентами. Отметим, что узлы первого уровня, которые служат для агрегирования результатов поиска, выглядят как обычные клиенты для узлов второго уровня, где, собственно, и производится поиск. Поэтому сначала обновляются узлы второго уровня, потом — первого уровня, а в последнюю очередь — приложение на веб-сервере.

Вот какие уроки мы извлекли из этого примера.

- Девиз разработчика очень большой базы данных: «Секционируй, секционируй, секционируй, распараллеливай».
- На крупных поисковых фермах организуйте демоны searchd в виде древовидных иерархий с несколькими уровнями.

- По возможности стройте оптимизированные индексы, каждый из которых содержит только часть данных.
- Устанавливайте соответствие между файлами и дисками явно, не полагаясь на RAID-контроллер.

Оптимизация выборки на сайте Sahibinden.com

У ведущего турецкого аукционного сайта Sahibinden (http://www.sahibinden.com) было немало проблем с производительностью, в том числе из-за полнотекстового поиска. После развертывания Sphinx и профилирования некоторых запросов выяснилось, что Sphinx способна обрабатывать многие часто выполняемые запросы с фильтрами быстрее, чем MySQL, несмотря на то что по одному из участвующих столбцов был построен индекс. Кроме того, использование Sphinx для обычного (не полнотекстового) поиска позволило унифицировать код приложения так, что его стало проще писать и сопровождать.

MySQL не показывала высокой производительности, поскольку селективность индексов по отдельным столбцам была недостаточной для существенного сокращения пространства поиска. Более того, было практически невозможно создать и поддерживать все необходимые индексы, так как в условиях отбора участвовало слишком много столбцов. В таблице лотов было около 100 столбцов, и теоретически каждый из них мог применяться для фильтрации или сортировки.

Интенсивная вставка и обновление горячих лотов выполнялись с черепашьей скоростью, так как приходилось обновлять слишком много индексов.

Поэтому система Sphinx стала естественным выбором для выполнения всех, а не только полнотекстовых запросов `SELECT` к таблице лотов.

Приведем размер базы данных и характеристики нагрузки.

- База данных содержит примерно 400 000 записей, а ее объем составляет 500 Мбайт.
- В день выполняется около 3 миллионов запросов.

Чтобы эмулировать обычные запросы `SELECT` с условиями `WHERE`, в полнотекстовый индекс Sphinx были добавлены специальные ключевые слова. Они имели вид `_ _CAT`*`N`*`_ _ _`, где вместо *`N`* подставлялся идентификатор соответствующей категории. Это делалось на этапе индексирования путем включения функции `CONCAT()` в запрос к MySQL, поэтому изменять источник данных не пришлось.

Индексы нужно было перестраивать как можно чаще. Мы решили делать это раз в минуту. На полную перестройку индекса уходило 9–15 секунд на одном из многих процессоров, поэтому схема «главный + разностный», которую мы обсуждали ранее, оказалась излишней.

Как выяснилось, API PHP тратил довольно много времени (7–9 миллисекунд на запрос) на разбор результирующего набора, в котором было много атрибутов. Обычно эти расходы можно не принимать во внимание, так как затраты на полнотекстовый

поиск, особенно в больших наборах данных, безусловно, доминируют. Но в данном случае необходимо было также выполнять неполнотекстовые запросы к небольшому набору данных. Чтобы решить эту проблему, мы разбили индексы на пары, состоящие из облегченного, который содержал 34 наиболее востребованных атрибута, и полного, содержавшего все 99 атрибутов.

Можно было бы поступить и по-другому: воспользоваться подсистемой SphinxSE или перенести в Sphinx только часть столбцов. Однако реализовать решение на базе двух индексов оказалось быстрее, а время играло решающую роль.

Из этого примера мы извлекли следующие уроки.

- Иногда полное сканирование средствами Sphinx работает быстрее, чем доступ по индексу MySQL.
- Для условий с высокой селективностью применяйте фиктивные ключевые слова вместо фильтрации по атрибутам, тогда подсистема полнотекстового поиска сможет взять на себя бóльшую часть работы.
- API скриптовых языков может оказаться узким местом в некоторых нетипичных, но встречающихся на практике случаях.

Оптимизация GROUP BY на сайте BoardReader.com

Для улучшения службы BoardReader потребовалось подсчитывать гиперссылки и строить по этим данным различные отчеты. Например, в одном из отчетов нужно было показать *N* верхних доменов второго уровня, которые сослались на BoardReader.com в течение прошлой недели. В другом отчете отображались *N* верхних доменов второго и третьего уровня, ссылающихся на данный сайт, например YouTube. Запросы, необходимые для построения этих отчетов, обладали следующими общими характеристиками.

- Они всегда группировали по домену.
- Они сортировали по количеству записей в группе или по количеству различных значений в группе.
- Они обрабатывали очень большой объем данных (миллионы записей), но результирующий набор, состоящий из лучших групп, был мал.
- Были допустимы приближенные результаты.

На этапе тестирования прототипа выяснилось, что MySQL тратит на выполнение таких запросов до 300 секунд. Теоретически путем секционирования данных, размещения их на разных серверах и ручного агрегирования результатов в приложении можно было довести время выполнения до 10 секунд. Но подобную архитектуру трудно построить — даже одна лишь реализация секционирования далеко не тривиальна.

Поскольку мы добились успеха в применении Sphinx для распределения поисковой нагрузки, решено было использовать Sphinx и в целях реализации приближенной группировки. Для этого потребовалось обработать данные перед индексированием,

преобразовав все интересующие нас подстроки в отдельные «слова». Вот пример URL до и после предобработки:

```
source_url      = http://my.blogger.com/my/best-post.php
processed_url   = my$blogger$com, blogger$com, my$blogger$com$my,
                  my$blogger$com$my$best, my$blogger$com$my$best$post.php
```

Знаки доллара (`$`) просто заменяют все символы-разделители в URL. Необходимо это для того, чтобы поиск можно было выполнять по любой части URL, будь то доменное имя или путь. В результате предварительной обработки все интересные подстроки выделяются в виде обособленных слов, чтобы поиск протекал максимально быстро. Технически мы могли использовать для поиска целую фразу или применить индексирование по префиксу, но тогда индексы получились бы более громоздкими, а производительность снизилась бы.

Ссылки преобразуются во время индексирования с помощью специально написанной пользовательской функции (UDF). Для решения этой задачи мы модифицировали Sphinx так, чтобы она могла подсчитывать различные значения. В итоге мы смогли полностью перенести запросы на поисковый кластер, распределить их и тем самым существенно снизить время задержки.

Приведем размер базы данных и характеристики нагрузки.

- В базе 150–200 миллионов записей, после предварительной обработки это выливается в 50–100 Гбайт данных.
- Нагрузка составляет 60 000–100 000 запросов с группировкой в день.

Индексы для распределенных запросов `GROUP BY` были размещены на том же поисковом кластере из шести машин с 24 процессорами, который был описан ранее. Это лишь немного увеличило нагрузку по сравнению с той, что создается поисковыми запросами к базе данных объемом 1,5 Тбайт.

Sphinx позволила заменить точные, но медленные вычисления, выполняемые сервером MySQL на одном процессоре, приближенными, но быстрыми и распределенными. Присутствовали все факторы, вызывающие погрешности: входные данные часто содержали слишком много строк, которые не помещались в буфер сортировки (мы ограничили его размер — 100 000 строк), использовалась агрегатная функция `COUNT(DISTINCT)` и формировались результирующие наборы, полученные от разных узлов. Несмотря на это, итоги для первых 10–1000 групп (а именно столько нужно было для отчетов) оказались правильными на 99–100 %.

Индексированные данные сильно отличались от тех, что использовались бы при обычном полнотекстовом поиске. Количество документов и ключевых слов огромно, хотя сами документы совсем маленькие. Их нумерация непоследовательная, так как применяется специальное соглашение о присвоении номеров (сервер-источник, таблица-источник и первичный ключ) и результирующий номер не умещается в 32 бита. Огромное количество ключевых слов часто приводило к коллизиям при вычислении CRC32-свертки (в Sphinx используется алгоритм CRC32 для отображе-

ния самих ключевых слов на их внутренние идентификаторы). Из-за этого пришлось повсюду перейти к 64-разрядным идентификаторам.

В настоящий момент производительность удовлетворительная. Для самых сложных доменов запрос обычно завершается за 0,1–1,0 секунды.

Из этого примера мы извлекли следующие уроки.

- Для запросов с фразой `GROUP BY` точность иногда можно принести в жертву скорости.
- Для очень больших наборов текстовых документов и даже для наборов умеренного размера, но специального вида могут понадобиться 64-разрядные идентификаторы

Оптимизация запросов с JOIN к шардированным данным на сайте Grouply.com

На сайте Grouply (http://www.grouply.com) решение на основе Sphinx позволило осуществлять поиск в базе данных, насчитывающей многие миллионы тегированных сообщений. Для повышения масштабируемости эта база распределена между несколькими физическими серверами, поэтому иногда приходится опрашивать таблицы, находящиеся на разных машинах. Так как количество серверов, баз данных и таблиц велико, то выполнять произвольные запросы с соединениями невозможно.

Для хранения тегов сообщений на сайте Grouply применяются многозначные атрибуты Sphinx. Список тегов извлекается из кластера Sphinx с помощью API PHP. Это позволяет заменить несколько последовательных запросов `SELECT` к разным серверам. Чтобы уменьшить и количество SQL-запросов, некоторые данные, требующиеся только для отображения (например, короткий список пользователей, недавно прочитавших сообщение), также хранят в отдельном MVA и доступ к ним можно получить через Sphinx.

Два основных новшества, примененных на этом сайте, — это эксплуатация системы Sphinx для предварительного построения результатов соединения и использование ее распределенных возможностей для объединения данных, разбросанных по многим шардам. Сделать это с помощью одной лишь MySQL было бы почти невозможно. Для эффективного объединения нужно было стремиться к уменьшению количества серверов и таблиц, содержащих секции, но это вошло бы в противоречие с масштабируемостью и расширяемостью.

Из этого примера мы извлекли следующие уроки.

- Sphinx можно использовать для эффективного агрегирования глубоко секционированных данных.
- Многозначные атрибуты можно использовать для хранения и оптимизации предварительного построенных результатов операции `JOIN`.

Итоги

В этом приложении мы смогли лишь кратко рассмотреть систему полнотекстового поиска Sphinx. Мы сознательно опустили многие средства Sphinx, например индексирование HTML-документов, запросы с указанием диапазона для улучшенной поддержки таблиц типа MyISAM, поддержку морфологии и синонимии, индексирование по префиксам и инфиксам и индексирование документов на восточных языках. Тем не менее надеемся, что вы смогли получить представление о том, как с помощью Sphinx можно эффективно решать различные реальные задачи. Эта система не ограничивается полнотекстовым поиском, ее можно применить ко многим трудным задачам, которые традиционно решались с помощью SQL.

Sphinx не является ни панацеей, ни заменой MySQL. Однако во многих случаях (а их в современных веб-приложениях становится все больше) ее можно использовать как полезное дополнение к MySQL. Она не только позволяет разгрузить сервер базы данных, но и открывает новые возможности для создания приложений.

Скачайте Sphinx с сайта http://www.sphinxsearch.com и не забудьте поделиться своими идеями!

САЛД

САНКТ-ПЕТЕРБУРГСКАЯ
АНТИВИРУСНАЯ
ЛАБОРАТОРИЯ
ДАНИЛОВА

www.SALD.ru
8 (812) 336-3739

Антивирусные
программные продукты

Хотите выжать из MySQL максимум возможностей?

Вам поможет уникальная книга, написанная экспертами для экспертов.

Познакомьтесь с продвинутыми приемами работы с MySQL: разработкой схем, индексов и запросов для настройки сервера, операционной системы и аппаратной части, способами масштабирования приложений и репликацией, балансировкой нагрузки, обеспечением доступности и восстановлением после отказов.

Прочитав эту книгу, вы узнаете, почему MySQL устроена именно так, познакомитесь с разбором практичных кейсов, научитесь мыслить на одном языке с вашей базой данных.

Бестселлер Шварца, Зайцева и Ткаченко — книга, необходимая любому профессионалу и способная превратить самую страшную «нештатную ситуацию» в легко преодолимый «рабочий момент».

Читайте и совершенствуйтесь!

Базы данных и системы управления базами данных (СУБД) Уровень пользователя: **опытный**

ПИТЕР®

Заказ книг:
тел.: (812) 703-73-74
books@piter.com

WWW.PITER.COM
каталог книг
и интернет-магазин

vk.com/**piterbooks**
instagram.com/**piterbooks**
facebook.com/**piterbooks**
youtube.com/**ThePiterBooks**

ISBN 978-5-4461-0696-7