



Быстрое решение возникающих проблем

Администрирование PostgreSQL 9

Книга рецептов

Как решать реальные задачи администрирования PostgreSQL:
более 100 простых, но невероятно эффективных рецептов

Саймон Ригс

Ханну Кросинг



Саймон Ригс
Ханну Кросинг

Администрирование PostgreSQL 9. Книга рецептов



Москва, 2013

PostgreSQL 9 Administration Cookbook

Solve real-world PostgreSQL problems
with over 100 simple, yet incredibly effective recipes

Simon Riggs
Hannu Krosing

[PACKT] open source 
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

Администрирование PostgreSQL 9 Книга рецептов

**Как решать реальные задачи администрирования PostgreSQL:
более 100 простых, но невероятно эффективных рецептов**

Издание рекомендовано в качестве учебного пособия
для студентов технических вузов

**Саймон Ригс
Ханну Кросинг**



УДК 004.6PostgreSQL 9
ББК 32.973-018.2
P49

Саймон Ригг, Ханну Кроснинг

P49 Администрирование PostgreSQL 9. Книга рецептов. Пер. с англ.: Самохвалова Е.В. – М.: ДМК Пресс, 2012. – 368 с.: ил.

ISBN 978-5-94074-750-5

Эта книга поможет вам освоить систему управления базами данных PostgreSQL. Здесь вы найдете рекомендации ведущего разработчика функционала репликации и восстановления PostgreSQL и системного архитектора Skype – одного из самых успешных проектов, созданных с помощью PostgreSQL. Данное руководство должно стать настольным для разработчиков, имеющих дело с «боевыми» базами данных и использующих PHP, Java, Python, Ruby, .Net в любой среде разработки.

В книге представлены ответы на общие вопросы и решения проблем, почерпнутые авторами из личного опыта работы с СУБД PostgreSQL – в качестве пользователей, разработчиков и преподавателей.

Каждая задача разбивается на простые части, или рецепты, проиллюстрированные кодом и сопровождаемые подробными объяснениями, как и почему это работает. Таким образом, руководство будет полезно как новичкам, так и специалистам. Кроме того, здесь отражены все новые возможности PostgreSQL 9.

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-849510-28-8 (англ.)
ISBN 978-5-94074-750-5 (рус.)

© Published by Packt Publishing Ltd.
© Оформление, перевод на русский язык
ДМК Пресс, 2013

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по почтовому адресу: 123242, Москва, а/я 20 или по электронному адресу: orders@alians-kniga.ru.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: www.alians-kniga.ru.

Оптовые закупки: тел. (499) 725-54-09, 725-50-27; электронный адрес books@alians-kniga.ru.

Саймон Ригс
Ханну Кросинг

Администрирование PostgreSQL 9. Книга рецептов

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru

Корректор *Готлиб О. В.*

Верстка *Татаринов А. Ю.*

Дизайн обложки *Мовчан А. Г.*

Подписано в печать 19.10.2012. Формат 60×90 1/16.

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 22,75. Тираж 100 экз.

Веб-сайт издательства: www.dmk-press.ru

POSTGRESMEN

Компания «Постгресмен» (<http://postgresmen.ru>) с 2007 предоставляет услуги по настройке производительности, поддержке, администрированию баз данных PostgreSQL в крупных проектах, обслуживающих десятки миллионов пользователей онлайн. Компания сотрудничает с разработчиками ядра PostgreSQL, ежегодно представляя широкой публике возможность пообщаться со звездами мировой величины – разработчиками ядра СУБД PostgreSQL, а также техническими специалистами крупнейших проектов со всего мира.

«Обмен опытом с экспертами мирового уровня – необходимое условие для того, чтобы мы могли говорить о том, что наша IT-индустрия, особенно та ее часть, что опирается на продукты Open Source, качественно росла из года в год, – считает основатель компании «Постгресмен» Николай Самохвалов. – Мы искренне рады, что в 2012 году на конференции Highload++ с ее тысячей посетителей нам с коллегами из PostgreSQL-Consulting.com удалось организовать целый день интереснейших докладов по PostgreSQL. А выход данной книги – еще один значительный шаг для популяризации этой замечательной системы в русскоязычной среде».

Содержание

Об авторах.....	12
О рецензентах	13
Предисловие	14
О чем эта книга.....	14
Что вам понадобится	17
Для кого написана книга.....	17
Условные обозначения	18
Сообщение об ошибках	18
Незаконное воспроизведение содержимого книги.....	19
Вопросы	19
ГЛАВА 1.	
Первые шаги.....	20
Введение.....	20
Представляем PostgreSQL 9	20
Где взять PostgreSQL	23
Соединение с сервером PostgreSQL	25
Открытие доступа для сетевых/удаленных пользователей.....	28
Использование графических инструментов администрирования	30
Использование инструмента для написания запросов и скриптов psql	34
Безопасная смена пароля	38
Как избежать хранения пароля в коде.....	39
Использование сервисного файла соединений	40
Если установить соединение не удастся.....	41
ГЛАВА 2.	
Исследование базы данных.....	44
Введение.....	44
Как узнать версию сервера.....	44
Определение времени работы сервера	46
Где находятся файлы сервера БД	47

Где находится журнал сообщений сервера	49
Где находится системный идентификатор БД	52
Список баз данных на сервере	53
Сколько таблиц в базе данных	56
Сколько места на диске занимает база данных	59
Сколько места на диске занимает таблица	59
Какие таблицы являются самыми большими	60
Сколько строк в таблице	61
Быстрый подсчет строк в таблице	62
Понятие объектных зависимостей	67

ГЛАВА 3.

Конфигурация 70

Введение	70
«Правильная» документация (RTFM)	70
Планирование новой базы данных	72
Изменение параметров в программах	73
Текущие конфигурационные настройки	75
Параметры, не определяемые по умолчанию	76
Обновление файла параметров	78
Настройка параметров для определенных групп пользователей	79
Основной список параметров для проверки конфигурации сервера	80
Добавление внешних модулей в PostgreSQL	82
Запуск сервера в энергосберегающем режиме	84

ГЛАВА 4.

Контроль над сервером 86

Введение	86
Запуск сервера БД вручную	87
Быстрая и безопасная остановка сервера	88
Экстренная остановка сервера	89
Перезагрузка конфигурационных файлов	89
Быстрый перезапуск сервера	91
Запрет на создание новых соединений	93
Создание ограничения по одной сессии на пользователя	94
Принудительное отключение пользователей	96
Проектирование с учетом multi-tenancy	97
Использование множественных схем	99
Выделение для пользователя собственной базы данных	101
Запуск нескольких серверов на одной системе	103
Настройка пула соединений	104

ГЛАВА 5.**Таблицы и данные 108**

Введение	108
Выбор имени для объекта БД	108
Управление объектами с именами, заключенными в кавычки	110
Принудительная установка единого имени, единого определения столбцов	112
Обнаружение и удаление дубликатов	116
Предотвращение появления строк-дубликатов	119
Поиск уникального ключа для выборки данных	124
Генерирование тестовых данных	126
Случайные данные по примеру	129
Подгрузка данных из электронных таблиц	132
Подгрузка данных из файлов	135

ГЛАВА 6.**Безопасность 139**

Введение	139
Как забрать у пользователя права доступа к таблице	140
Как предоставить пользователю доступ к таблице	142
Создание нового пользователя	144
Временный запрет соединений для пользователя	145
Удаление пользователя без потери его данных	146
Проверка безопасности паролей пользователей	148
Назначение определенным пользователям ограниченных прав суперпользователя	150
Аудит изменений DDL	153
Аудит изменений данных	154
Интеграция с LDAP	158
Соединение с использованием SSL	159
Шифрование секретных данных	162

ГЛАВА 7.**Администрирование баз данных 166**

Введение	166
Написание скрипта, который срабатывает полностью или не работает вообще	167
Написание скрипта psql, который прерывается после первой ошибки	170
Действия с несколькими таблицами	171
Добавление и удаление столбца	176

Изменение типа данных столбца	178
Добавление и удаление схем.....	181
Перемещение объектов между схемами.....	183
Добавление и удаление табличных пространств.....	184
Перемещение объектов между табличными пространствами....	188
Доступ к объектам, находящимся в других БД PostgreSQL.....	191
Обновление представления	196

ГЛАВА 8.

Мониторинг и диагностика 202

Введение.....	202
Подключен ли пользователь	205
Что выполняет пользователь	207
Активен или заблокирован запрос	209
Почему запрос заблокирован.....	211
Как прервать сессию	212
Принятие решения о сомнительной подготовленной транзакции	215
Использует ли кто-либо определенную таблицу	215
Когда в последний раз использовали таблицу	217
Сколько места на диске будут занимать временные данные	220
Почему запросы выполняются медленно	222
Обнаружение ошибок и сообщение о них	225
Подведение итога записей в журнале об ошибках за день	227

ГЛАВА 9.

Регулярное обслуживание 230

Введение.....	230
Управление автоматическим обслуживанием базы данных ...	231
Предотвращение «автозаморозки» и повреждения страниц.....	236
Предупреждение «свертывания» счетчика транзакций	238
Удаление ранее подготовленных транзакций.....	240
Для любителей временных таблиц.....	243
Обнаружение и исправление раздутых таблиц и индексов.....	244
Обслуживание индексов.....	249
Обнаружение неиспользуемых индексов	252
Безопасное удаление ненужных индексов	254
Планирование обслуживания	255

ГЛАВА 10.

Производительность и согласованность 258

Введение.....	258
Как найти медленные запросы SQL	259

Как получить регулярную статистику из представлений <code>pg_stat*</code>	262
Как установить причины замедления работы SQL	263
Как уменьшить число возвращаемых строк	268
Как упростить сложный SQL	269
Как ускорить выполнение запросов, не переписывая их	275
Почему запрос не использует индекс	279
Как заставить запрос использовать индекс	280
Использование оптимистичных блокировок	282
Сообщения о проблемах с производительностью	284

ГЛАВА 11.

Резервное копирование и восстановление 285

Введение	285
Что такое восстановление после сбоя и как его контролировать	286
Планирование резервного копирования	288
Горячее логическое резервное копирование одной базы данных	291
Горячее логическое резервное копирование всех баз данных	293
Горячее логическое резервное копирование всех таблиц из табличного пространства	294
Резервное копирование определений объектов базы данных	296
Автономное горячее физическое резервное копирование базы данных	297
Горячее физическое резервное копирование и непрерывное архивирование	300
Восстановление всех баз данных	303
Восстановление до определенной временной точки	307
Восстановление удаленной или поврежденной таблицы	310
Восстановление удаленного или поврежденного табличного пространства	312
Восстановление удаленной или поврежденной базы данных	314
Улучшение производительности резервного копирования и восстановления	315
Инкрементальное и дифференциальное резервное копирование и восстановление	318

ГЛАВА 12.

Репликация и обновления	321
Репликация: основные термины	322
Лучшие способы репликации.....	326
Файловая репликация с передачей журналов.....	328
Настройка потоковой журнальной репликации	332
Управление репликацией с передачей журналов	337
Управление режимом горячего резервирования	341
Выборочная репликация с помощью Londiste 3.0.....	346
Выборочная репликация с помощью Slony 2.0	351
Балансировка нагрузки с помощью pgpool-II 3.0.....	356
Минорное обновление.....	359
Мажорные обновления на месте.....	360
Мажорные онлайн-обновления с помощью инструментов репликации	362

Об авторах

Саймон Ригс является ведущим разработчиком проекта PostgreSQL, а также техническим директором компании 2ndQuadrant, предоставляющей круглосуточную поддержку пользователям PostgreSQL по всему миру.

Саймон внес значительный вклад в развитие PostgreSQL, поскольку его работа напрямую связана с построением архитектуры баз данных и решением возникающих проблем. Среди добавленных им возможностей можно назвать восстановление точки во времени (Point in Time Recovery), репликацию с режимом с серверами теплого резерва (Warm Standby replication), режим горячего резерва (Hot Standby), асинхронную фиксацию транзакций (Asynchronous Commit), партиционирование (Partitioning). Его решения, позволяющие повысить производительность и усовершенствовать настройку, в огромном количестве встречаются в коде и документации по PostgreSQL.

Саймон также работал в Oracle, Teradata и DB2, имеет множество сертификатов. В качестве технического руководителя он трудился в банковской сфере, области телекоммуникаций и разработки программного обеспечения. Одно из его исследований было опубликовано Королевским обществом.

Ханну Кросинг является старшим консультантом в 2ndQuadrant и техническим советником в Ambient Sound Investments. В компании Skype Technologies он был архитектором баз данных и проектировал модуль для репликации Skytools и методы масштабирования. Уже 12 лет является разработчиком PostgreSQL.

О рецензентах

Габриэль Бартолини, опытный разработчик открытого программного обеспечения, уже более десяти лет создает приложения для Linux/Unix на C и C++ и специализируется на поисковых машинах и веб-аналитике в больших базах данных.

Габриэль защитил диссертацию по статистике в Университете Флоренции. Анализируя веб-трафик в Австралии и Италии, он приобрел большой опыт в вопросах сбора, анализа и хранения данных.

Габриэль является консультантом в компании 2ndQuadrant и активным членом международного сообщества PostgreSQL.

В настоящее время Габриэль живет в Прато, небольшом, но оживленном городке на севере Тосканы в Италии. Вторым домом он считает австралийский Мельбурн, где он учился в Университете Монаша и работал в области ИТ.

Увлекается футболом и игрой на электрогитаре.

Благодарю свою семью, особенно Кэти, которая постоянно подбадривала меня, напоминая, что учиться никогда не поздно.

Дмитрий Фонтейн входит в сообщество PostgreSQL и пишет программы с открытым кодом уже более десяти лет. Он разрабатывал такие промышленные решения, как `pgloader` (ETL), `PostgreSQL prefix indexing` (телефонная маршрутизация), `pg_staging` (поддержка окружения из резервных копий), `preprepare` (позволяет легко использовать подготовленные выражения в пулере соединений (`connection pooler`)) и патчи для обновлений и дополнений к PostgreSQL.

Кроме того, он принимал участие в разработке Skytools и системы репликации Londiste и создал поддержку PHP для системы организации асинхронной работы с очередями событий PgQ.

Его конек – улучшение системной архитектуры и дизайна с целью сокращения времени обслуживания и повышения доступности.

В настоящее время Дмитрий работает старшим консультантом в компании 2ndQuadrant во Франции. Профессиональный опыт и отличное знание PostgreSQL позволяют ему вести активную работу в сообществе.

Предисловие

PostgreSQL – это продвинутый сервер баз данных, который доступен на множестве платформ. Заработав отличную репутацию благодаря высокой производительности, стабильности и широким возможностям, PostgreSQL быстро набирает популярность, становясь одной из самых широко используемых СУБД в мире. Один из первых проектов с открытым кодом, в котором участвует множество разработчиков со всего мира, является абсолютно бесплатным. И это работает!

Преимущества свободного программного обеспечения с открытым кодом очевидны: вы вольны устанавливать, использовать и распространять PostgreSQL, и за это не нужно никому платить. Главное же заключается в том, что СУБД PostgreSQL способна долго и бесперебойно работать при минимальной поддержке со стороны администратора. В общем, стоимость использования PostgreSQL довольно мала.

Эта книга поможет вам управлять рабочими базами данных PostgreSQL. Здесь вы найдете рекомендации ведущего разработчика функционала репликации и восстановления PostgreSQL и системного архитектора Skype – самого успешного стартапа, созданного с помощью PostgreSQL. Данное руководство должно стать настольным для разработчиков, имеющих дело с «боевыми» базами данных и использующих Java, Python, Ruby, .Net в любой среде разработки.

В книге представлены ответы на общие вопросы и решения проблем, почерпнутые авторами из личного опыта работы с СУБД PostgreSQL – в качестве пользователей, разработчиков и преподавателей.

Каждая задача разбивается на простые части, или рецепты, проиллюстрированные кодом и сопровождаемые подробными объяснениями, как и почему это работает. Таким образом, руководство может быть полезным как новичкам, так и специалистам. Кроме того, здесь отражены все новые возможности PostgreSQL 9.

О чем эта книга

В первой главе вы познакомитесь с PostgreSQL 9, узнаете, как скачать и установить PostgreSQL, как подключиться к серверу PostgreSQL, открыть доступ для сетевых/удаленных пользователей, как использовать визуальные инструменты администрирования, с помощью каких инструментов писать скрипты и запросы `psql`, как безопасно

менять свой пароль, использовать служебный файл соединений и выяснить причины потери соединения.

Вторая глава поможет вам определить версию сервера БД и время его работы. Вы поймете, где находятся файлы сервера БД, журнал сообщений сервера и системный идентификатор СУБД. Вы сможете просмотреть список баз данных на сервере и узнаете, сколько таблиц в базе данных, сколько места на диске занимают БД и отдельные таблицы, какие таблицы являются самыми большими, сколько строк в таблице и как определить зависимости объектов.

В третьей главерассматривается, как читать документацию (RTFM), планировать новую базу данных, изменять параметры в программах, как узнавать текущие конфигурационные настройки, параметры, которые не являются значениями по умолчанию, как обновлять файл параметров, как устанавливать параметры для определенных групп пользователей. Также описываются простой список для проверки конфигурации сервера, добавление на сервер PostgreSQL внешних модулей и запуск сервера в режиме экономии энергии.

В четвертой главе вы найдете информацию о ручном запуске сервера БД, быстрой и безопасной остановке сервера, остановке сервера в экстренных случаях, перезагрузке конфигурационных файлов сервера, быстром перезапуске сервера, запрете на установку новых соединений, установке ограничения по одной сессии на каждого пользователя и отключении пользователей от системы. Простые рецепты помогут вам организовать работу в команде, применять множественные схемы, выделять для пользователей приватные базы данных, запускать несколько серверов БД на одной системе и настраивать пул соединений.

Из пятой главы вы узнаете, как выбирать имена для объектов баз данных, как управлять объектами с квотированными именами, присваивать одно имя, одно определение для разных столбцов, обнаруживать и удалять строки-дубликаты, предотвращать появление строк-дубликатов, как находить уникальный ключ для набора данных, генерировать тестовые данные, случайные данные, подгружать данные из динамических таблиц и из неструктурированных файлов.

В шестой главе содержатся рецепты восстановления доступа пользователя к таблице, предоставления пользователю прав доступа к таблице, создания нового пользователя, временного ограничения соединения для пользователя, удаления пользователя без потери его данных, проверки безопасности пароля пользователей, делегирования ограниченных прав суперпользователя для определенных

пользователей, аудита изменений DDL, аудита изменений данных, интеграции с LDAP, установки соединения с использованием SSL и шифровании важных данных.

В седьмой главе рассказывается, как писать скрипт, в котором все строки должны выполняться или, наоборот, не выполняться, как создать скрипт в `psql`, который вызывается при первой ошибке, как выполнять действия сразу с несколькими таблицами, добавлять/удалять столбцы, изменять тип данных столбца, добавлять/удалять схемы, перемещать объекты между схемами, добавлять/удалять табличные пространства, перемещать объекты между табличными пространствами, получать доступ к объектам из других баз данных PostgreSQL и делать представления обновляемыми.

Рецепты, содержащиеся в восьмой главе, позволяют понять, подключен ли пользователь, что он запускает, является ли он активным или его заблокировали, кто его заблокировал, использует ли кто-либо определенную таблицу и когда он в последний раз это делал, сколько места на диске занимают временные данные, почему ваши запросы замедляются. Кроме того, вы узнаете, как обнаружить ошибку и сообщить о ней, как создавать ежедневный итоговый отчет на основе журнализации ошибок, как прервать определенную сессию и что делать с сомнительной подготовленной транзакцией.

Из девятой главы вы узнаете, как контролировать автоматическую поддержку базы данных, как избежать «автозаморозки» и повреждения страниц, обертывания транзакций, как удалять старые подготовленные транзакции, действия для «тяжелых»¹ пользователей со временными таблицами, как обнаруживать и исправлять раздутые таблицы и индексы, как поддерживать индексы, находить неиспользуемые индексы, аккуратно удалять ненужные индексы и планировать обслуживание.

В десятой главе описано, как находить медленные SQL-параметры, получать регулярную статистику из представлений `pg_stat*`, выявлять причины замедления SQL, уменьшать количество возвращаемых строк, упрощать сложный SQL, ускорять запросы без переписывания. Вы узнаете, почему запрос не использует индекс и как заставить запрос его задействовать. Кроме того, рассматривается использование оптимистичных блокировок и сообщение о проблемах с производительностью.

В одиннадцатой главе описываются резервное копирование и восстановление. Многие признают важность резервного копирования, но мало кто думает об этом всерьез. Из этой главы вы получите важную

¹ Речь идет о пользователях, генерирующих «тяжелые» (сильно нагружающие систему) запросы и/или о пользователях с плохим каналом связи. — *Прим. ред.*

информацию о том, как создавать резервную копию и как восстанавливать из нее базу данных PostgreSQL, узнаете о планировании резервного копирования, горячем логическом резервном копировании как одной, так и всех баз данных, горячем логическом резервном копировании всех таблиц в табличном пространстве, резервном копировании определенных объектов БД, отдельном горячем физическом резервном копировании БД, горячем физическом резервном копировании и непрерывном архивировании. Кроме того, в этой главе рассматривается восстановление всех баз данных, восстановление к какому-либо моменту времени, восстановление удаленной или поврежденной таблицы, табличного пространства, улучшение производительности резервного копирования/восстановления и инкрементальное/дифференциальное резервное копирование и восстановление.

Двенадцатая глава посвящена репликации и обновлениям. Репликация – это не волшебство, но вещь удивительная (еще удивительнее, когда она работает). В этой главе дается определение репликации, описываются лучшие приемы, настройка репликации, основанной на трансфере файлов журнала, управление режимом hot standby, выборочная репликация с использованием Londiste 3.0, выборочная репликация с использованием Slony 2.0, балансирование загрузки с помощью pgpool II 3.0, обновление до нового минорного релиза (например, с 9.0.0 до 9.0.1), мажорные обновления на месте (например, с 8.4 на 9.0 или с 9.0 на 9.1) и мажорные обновления онлайн с помощью инструментов репликации.

Что вам понадобится

Для работы с книгой вам понадобится следующее программное обеспечение:

- PostgreSQL 9.0 Server Software;
- psql client utility (part of 9.0);
- pgAdmin3 1.12.

Для кого написана книга

Руководство предназначено системным администраторам, администраторам баз данных, архитекторам, разработчикам, а также всем, кто планирует работать с базами данных или уже занимается этим. В первую очередь она подойдет тем, у кого уже есть какой-либо технический опыт.

Условные обозначения

Кодовые слова в тексте выделены моноширинным шрифтом. Например: «В PostgreSQL 9.0 вам больше не понадобится утилита `pg_standby`, поскольку многие ее функции теперь выполняет непосредственно сервер».

Блоки кода выглядят так:

```
CREATE USER repuser
    SUPERUSER
    LOGIN
    CONNECTION LIMIT 1
    ENCRYPTED PASSWORD 'changeme';
```

Фрагменты кода, на которые требуется обратить особенное внимание, выделены **полужирным моноширинным шрифтом**:

```
SELECT *
FROM mytable
WHERE (col1, col2, ... ,colN) IN(SELECT col1, col2, ... ,colN
FROM mytable
GROUP BY
col1, col2, ... ,colN HAVING count(*) > 1);
```

Все, что вводится или выводится в командной строке, также записывается **полужирным моноширинным шрифтом**:

```
$ postgres --single -D /full/path/to/datadir postgres
```

Новые термины и важные слова выделены *курсивом*.

Слова, которые вы можете видеть на экране, например в меню или диалоговых окнах, в тексте выделены следующим образом: «В Query есть замечательная опция **Visual Explain** и **Graphical Query Builder**».

Предупреждения, важные замечания, советы и подсказки вынесены в такие блоки.

Сообщение об ошибках

Хотя мы делаем все возможное для того, чтобы не допустить ошибок в наших изданиях, вы можете встретить в тексте какие-либо неточности. В таком случае мы будем рады, если вы сообщите нам об этом. Пожалуйста, зайдите на веб-страницу <http://www.packtpub.com/support>, выберите название книги, в которой вам встретилась ошибка, и опишите ее, заполнив соответствующую форму. Как только эта информация будет проверена, мы опубликуем ее в разделе «Errata» (Ошибки) на сайте. Уже найденные ошибки можно просмотреть,

выбрав название интересующей вас книги в разделе <http://www.packtpub.com/support>.

Сообщения об ошибках в русскоязычном издании книги можно оставить на сайте издательства «ДМК Пресс»:

<http://www.dmk-press.ru/contacts1/contacts/>

Незаконное воспроизведение содержимого книги

Пиратские копии – повсеместная проблема. Если вам встретились незаконным образом растиражированные экземпляры данной книги (в любом формате), пожалуйста, сообщите источник публикации, написав письмо по адресу dm@dmk-press.ru или copyright@packtpub.com.

Вопросы

Вы можете присылать любые вопросы, касающиеся данной книги, по адресу dm@dmk-press.ru и questions@packtpub.com. Мы постараемся решить возникшие проблемы.



ГЛАВА 1.

Первые шаги

Рассматриваемые темы:

- Представляем PostgreSQL 9
- Где взять PostgreSQL
- Соединение с сервером PostgreSQL
- Открытие доступа для сетевых/удаленных пользователей
- Использование графических инструментов администрирования
- Использование инструмента для написания запросов и скриптов psql
- Безопасная смена пароля
- Как избежать хранения пароля в коде
- Использование сервисного файла соединений
- Если установить соединение не удастся

Введение

PostgreSQL – это мощная система управления базами данных (СУБД), используемая для выполнения самых разных задач. Путь к овладению ею непрост, но любое путешествие должно начинаться с первого шага.

Для начала рассмотрим самое первое соединение с базой. Для многих оно оказывается непреодолимым препятствием, поэтому разберем его более подробно: сначала остановимся на доступе для удаленных пользователей, а оттуда доберемся до инструментов администрирования с графическим интерфейсом.

Также проанализируем инструмент запросов psql, который будет использоваться для загрузки нашей базы данных.

Кроме того, здесь представлено несколько полезных рецептов, которые пригодятся вам в будущем.

Представляем PostgreSQL 9

PostgreSQL – это продвинутый сервер для баз данных SQL, доступный на множестве платформ.

Одно из очевидных преимуществ СУБД PostgreSQL заключается в ее открытости: вы можете совершенно бесплатно устанавливать, использовать и распространять ее. Но самое главное – PostgreSQL славится своей стабильностью, и во многих случаях необходима лишь минимальная поддержка. Эта система очень недорога.

Возможности PostgreSQL огромны, ведь она разрабатывается и улучшается на протяжении вот уже 20 лет – начиная с рождения в Группе исследований проблем баз данных Калифорнийского университета Беркли. Сегодня свой вклад вносит огромная армия разработчиков кода, архитекторов, администраторов БД и преподавателей – многие из них используют PostgreSQL на своем основном месте работы. Некоторые, в том числе и авторы этой книги, работают в компаниях, специализирующихся на создании сервисов для PostgreSQL. Но нет такой компании, которая могла бы считать себя владельцем этой СУБД, и никто не требует и не предполагает, что вы где-то будете указывать, что используете PostgreSQL.

Ключевые особенности PostgreSQL:

- высокий уровень соответствия стандартам языка SQL вплоть до стандарта SQL:2008;
- архитектура «клиент-сервер»;
- рассчитанная на большое количество параллельных операций архитектура: чтение и запись в базу данных не мешают друг другу;
- большие возможности конфигурирования и работы с различными типами приложений;
- отличная масштабируемость и производительность в сочетании с широкими возможностями настройки.

Чем отличается PostgreSQL от других СУБД

При разработке PostgreSQL ставились следующие задачи:

- создание надежного высококачественного программного кода, сопровождаемого подробными комментариями;
- сведение к минимуму необходимости в обслуживании системы;
- соответствие стандартам SQL, интероперабельность, обратная совместимость кода;
- высокая производительность, безопасность и доступность.

Многих удивляет, что по функциональности PostgreSQL ближе к Oracle или Microsoft SQL Server, чем к MySQL. Да и связывает MySQL с PostgreSQL лишь то, что оба эти проекта обладают открытым исходным кодом, в то время как основные свойства и философия различаются в корне.

Одной из ключевых особенностей Oracle – начиная с Oracle 7 – была «изоляция снимков» (snapshot isolation), позволявшая читающим и пишущим процессам не блокировать друг друга, а работать независимо. Как ни удивительно, PostgreSQL – первая СУБД, в которой эта особенность заложена изначально. В PostgreSQL она называется MVCC (Multi-Version Concurrency Control – мультиверсионная система управления параллельным выполнением операций). Далее вы сможете ознакомиться с ней подробнее.

PostgreSQL – это СУБД для широкого использования. Вы настраиваете базу данных так, чтобы вам было удобно с ней работать. PostgreSQL представляет для этого огромное количество возможностей. Вы можете использовать как модель нормализованной базы данных, так и расширения для работы с массивами и record-подтипами, или же положиться на полностью динамическую схему, используя расширение *hstore*. Кроме того, PostgreSQL позволяет создавать серверные функции на одном из десятка различных языков программирования.

Одно из полезных свойств PostgreSQL – расширяемость. Вы можете добавлять свои типы данных, операторы, типы индексов и функциональные языки. Например, можно переопределить поведение части системы и изменить выполнение команд, используя систему плагинов, или же добавить новый оптимизатор.

Все эти свойства открывают для архитекторов программного обеспечения огромные возможности – без лишних проблем создавать и поддерживать приложения на протяжении длительного времени. К сожалению, нельзя уместить в одну книгу все, что могло бы пригодиться разработчикам. Данное руководство посвящено администрированию, техническому обслуживанию и резервному копированию.

В самом начале, когда PostgreSQL была только системой для исследования в академических кругах, основные разработки сводились к добавлению новых впечатляющих функций. За последние 15 лет было переписано и усовершенствовано огромное количество строк кода, что сделало PostgreSQL одним из самых больших и стабильных серверов, готовых для оперативного использования.

Возможно, вы где-то читали, что PostgreSQL работает медленнее, чем Ваша Любимая СУБД (какой бы она ни была). Мы, авторы

этой книги, можем сообщить, что на протяжении последних шести лет трудились над улучшением производительности и наша команда добилась больших успехов в создании высокопроизводительного и масштабируемого сервера. Это дает проекту PostgreSQL большее пространство для развития.

Кто использует PostgreSQL? Среди крупных компаний можно назвать Apple, BASF, Genentech, IMDB.com, Skype, NTT, Yahoo и The National Weather Service. По данным Европейской комиссии, число загрузок PostgreSQL за год доходит до миллиона, и многие пользователи БД считают PostgreSQL достойной альтернативой их привычной СУБД.

Осталось упомянуть еще одну вещь. При рождении PostgreSQL получила имя Postgres, поэтому в некоторых ключевых частях проекта вы можете встретить именно это слово. Например, база данных по умолчанию называется postgres, и при установке программного обеспечения это слово часто выступает в качестве имени пользователя. Именно поэтому многие сокращают PostgreSQL до Postgres и считают эти два названия взаимозаменяемыми.

PostgreSQL читается «пост-грес-ку-эл», а Postgres – «пост-грес».

Некоторые используют еще и название Postgre, но это только вносит дополнительную путаницу. Двух названий вполне достаточно – пожалуйста, используйте только их!

Где взять PostgreSQL

PostgreSQL на 100 % является свободным программным обеспечением с открытым кодом.

Вы можете свободно использовать, изменять и распространять PostgreSQL любым удобным для вас способом. Лицензия на применение PostgreSQL является улучшенной версией лицензии *BSD* (*Berkeley Distribution Software*) и в настоящее время называется *TPL* (*The PostgreSQL License*).

Как это сделать

PostgreSQL уже используется во многих пакетах приложений, так что, возможно, она уже установлена на ваших серверах. Многие дистрибутивы Linux включают в себя PostgreSQL как часть основной системы, или же PostgreSQL содержится на диске с этими дистрибутивами.

Единственная проблема в том, что, скорее всего, прилагаемая версия PostgreSQL не окажется самой последней. Точнее, она будет

самой свежей на момент выхода дистрибутива операционной системы. Как правило, нет смысла держаться именно за эту версию – такой подход не дает какой-либо дополнительной стабильности, а более новые версии СУБД так же хорошо поддерживаются различными Linux-дистрибутивами.

Если у вас до сих пор нет копии дистрибутива PostgreSQL или имеющаяся версия уже не нова, вы можете скачать исходники или бинарные пакеты для различных операционных систем на сайте <http://www.postgresql.org/download/>.

Конкретный способ установки зависит от используемой вами платформы, и универсального рецепта здесь нет. Просто следуйте инструкциям, прилагаемым к вашей версии дистрибутива – этого будет достаточно. Мы сознательно не описываем здесь процесс установки, чтобы не исказить информацию, приведенную специально в помощь вам.

Если вы хотите получать сообщения об обновлениях и последних новостях по электронной почте, можно подписаться на рассылку, описывающую обновления от всех производителей, которые поддерживают PostgreSQL. Ежемесячно вам будет приходить несколько писем о новых версиях ядра PostgreSQL, приложениях, конференциях, а также и другая информация для пользователей. Поддерживать связь с разработчиками порой небесполезно.

Подробнее о данной рассылке можно узнать по адресу <http://archives.postgresql.org/pgsql-announce/>.

Как это работает

Многие удивляются: неужели все действительно бесплатно? Не придется ли в итоге платить? Кто стал бы трудиться даром?

Программные продукты с открытым кодом, такие как PostgreSQL, создаются сообществом разработчиков и пользователей. Из множества маленьких песчинок вырастает большая гора. Среди этих людей есть и любители, и профессионалы, ведь проектирование приложений часто бывает полезным не только разработчикам, но и их работодателям.

Хотите верьте, хотите нет – это просто работает!

И еще

Не забудьте, что PostgreSQL – это не только ядро. Множество веб-сайтов предлагают дополнения, расширения и инструменты для PostgreSQL. В блогах можно обнаружить массу полезной информации

и подсказок, которые могут помочь в работе. И, наконец, существуют компании профессионалов, которые в любой момент придут вам на помощь.

Соединение с сервером PostgreSQL

Как подключиться к PostgreSQL? Для многих с подключения начинается первое знакомство с PostgreSQL, так что постараемся сделать его приятным. Попутно разберем проблемы, которые могут возникнуть, и методы их решения. Помните, что соединение должно быть безопасным, так что постараемся защитить данные, к которым хотим получить доступ.

Прежде чем вы сможете работать с базой данных, необходимо подключиться к серверу, который откроет *сессию*.

Сессия рассчитана на длительный период времени, поэтому можно один раз подключиться, выполнить все необходимые запросы (которых может быть много) и, наконец, отключиться. Существуют некоторые накладные расходы в процессе подключения. Это можно заметить, если несколько раз подряд приходится подключаться/отключаться, и в таком случае есть смысл изучить пулы соединений (*connection pools*). Они позволяют быстро разбираться с предыдущими сессиями при каждом последующем подключении.

Подготовка

Итак, возьмем базу данных. Если вы не знаете, где она находится, это сильно затруднит работу. Баз данных может быть несколько; следует знать, какая именно вам нужна в данный момент, и иметь к ней соответствующие права доступа.

Как это сделать

Для подключения к PostgreSQL нужно определить следующие пять параметров:

- узел (*host*) или сетевой адрес;
- порт (*port*);
- имя базы данных;
- пользователь (*user*);
- пароль (или другие средства аутентификации).

Для соединения необходимо, чтобы по адресу `host` располагался сервер PostgreSQL, слушающий порт номер `port`. На этом сервере должны быть база данных `dbname` и пользователь `user`. Узел должен принимать соединения от вашего клиентского приложения (об этом читайте в разделе «Открытие доступа для сетевых/удаленных пользователей»), и необходимо пройти аутентификацию с использованием того метода, который требуется для данного сервера. Например, бесполезно вводить пароль на сервере, где используется другой механизм аутентификации.

Практически все интерфейсы PostgreSQL используют библиотеку `libpq`. При использовании `libpq` большая часть параметров соединения идентичны, поэтому обсудим их здесь и более не будем возвращаться к данному вопросу.

Если вы не определили эти параметры, ищем значения, заданные с помощью следующих переменных окружения:

- `PGHOST` или `PGHOSTADDR`;
- `PGPORT` (если ничего не указано, установите значение 5432);
- `PGDATABASE`;
- `PGUSER`;
- `PGPASSWORD` (вот это вообще-то не рекомендуем делать).

Если вы как-либо определили первые четыре параметра, но не определили пароль, будем искать файл с паролем (см. раздел ниже).

В некоторых интерфейсах PostgreSQL протокол «клиент-сервер» используется напрямую, так что значения по умолчанию могут быть другими. Существенных различий быть не должно, но лучше проверить точный синтаксис для вашего интерфейса.

Как это работает

PostgreSQL – это клиент-серверная база данных. Система, на которой она запущена, считается узлом (`host`). Можно получить удаленный доступ к серверу PostgreSQL по сети, но нужно указать имя узла (`hostname`) или IP-адрес (`hostaddr`). Например, если вы хотите подключить TCP/IP к своей системе, в качестве узла указывайте `localhost`. Как правило, лучше использовать соединение через Unix-сокеты, которое выполняется, если имя узла начинается с наклонной черты (`/`). Предполагается, что имя узла соответствует имени директории (по умолчанию `/tmp`).

В любой системе может быть несколько серверов баз данных. Каждый сервер слушает определенный сетевой порт (для каждого сервера в системе порт свой). По умолчанию для PostgreSQL используется порт 5432. Он зарезервирован для PostgreSQL, что официально за-

регистрировано в IANA. (Во многих *nix-серверах можно видеть его использование в файле `/etc/services`.) Если в системе несколько серверов, номер порта может служить уникальным идентификатором.

Иногда сервер баз данных называют кластером баз данных, так как PostgreSQL позволяет определить несколько баз данных на любом сервере. Каждый запрос на соединение должен содержать имя базы данных, и после установления соединения вы будете иметь дело только с объектами указанной базы данных.

Имя пользователя применяется для идентификации соединения. По умолчанию количество соединений для одного пользователя неограниченно, но есть способ ввести такое ограничение; он будет описан в одном из рецептов. В более поздних версиях СУБД понятия «*пользователь*» (user) и «*группа*» (group) объединили в понятие «*роль*» (role). Пользователи в прежнем понимании – это своего рода «логин-роли». Можно считать, что «логин-роль» – такая роль, которой можно назначить привилегию CONNECT. Несмотря на объединение понятий, во многих случаях можно встретить прежние термины («пользователь» и «группа»).

При каждом соединении должна проводиться аутентификация. Способ определяется сервером.

Для каждого установленного соединения в любое время возможна одна активная транзакция и одна полностью активная инструкция.

Каждый сервер может одновременно поддерживать ограниченное количество соединений, поэтому если сервер перегружен, то запрос на соединение может быть отклонен.

Проверка информации о соединении

Чтобы убедиться, что вы подключились к нужному серверу нужным способом, выполните все или некоторые из перечисленных команд:

```
SELECT inet_server_port();
```

(Порт, который слушает сервер.)

```
SELECT current_database();
```

(Текущая база данных.)

```
SELECT current_user;
```

(Идентификатор текущего пользователя.)

```
SELECT inet_server_addr();
```

(IP-адрес сервера, принявшего соединение.)

SQL-запроса, показывающего пароль, по понятным причинам не существует.

Еще может пригодиться

```
SELECT version();
```

Смотрите также

В процессе соединения есть еще много тонкостей. Часть из них рассматривается в шестой главе, часть описана в этой. Самую подробную информацию вы найдете в документации по серверу PostgreSQL.

Открытие доступа для сетевых/удаленных пользователей

PostgreSQL поставляется во многих сборках и пакетах, и в некоторых случаях из соображений безопасности удаленные соединения по умолчанию оказываются заблокированными.

Как это сделать

- Добавьте в `postgresql.conf` следующую строку (или отредактируйте соответствующую):

```
listen_addresses = '*'
```

- Чтобы разрешить доступ ко всем базам данных всем пользователям по паролю, запишите в файл `pg_hba.conf` первой строкой:

# TYPE	DATABASE	USER	CIDR-ADDRESS	METHOD
host	all	all	0.0.0.0/0	md5

Как это работает

Параметр `listen_addresses` определяет IP-адрес, соединения с которым будет ожидать сервер. Это позволяет иметь в системе несколько сетевых карт. Как правило, вы ждете соединения по всем доступным адресам, потому что используете значение `*`.

В файле `pg_hba.conf` содержится набор правил аутентификации для данного узла. Правила рассматриваются последовательно, пока одно из них не сработает или пока попытка соединения не будет отклонена методом `reject`.

Правило из примера означает, что для соединения любого пользователя с любой базой данных на любом IP-адресе для аутентификации потребуется md5-шифрованный пароль.

Type = `host` означает удаленное соединение.

Database = `all` значит «для всех баз данных». Другие имена воспринимаются в точности так, как записаны, за исключением случаев, когда перед именем стоит плюс (+), что означает групповую роль.

Кроме того, можно перечислить имена пользователей через запятую или дать ссылку на файл со списком пользователей с помощью символа @. Также можно задать значение `sameuser`; тогда правило будет считаться выполненным, если имена базы данных и пользователя будут совпадать.

`User = all` значит «для всех пользователей». Другие имена воспринимаются в точности так, как записаны, за исключением случаев, когда перед именем стоит плюс (+), что означает групповую роль. Кроме того, можно перечислить имена пользователей через запятую или дать ссылку на файл со списком пользователей с помощью символа @.

`CIDR-ADDRESS` состоит из двух частей: IP-адрес/маска подсети. Маска подсети задается числом первых битов IP-адреса. Таким образом, /0 означает 0 битов IP-адреса, то есть доступ будет открыт для всех IP-адресов. Например, если указано `192.168.0.0/24`, то берем первые 24 бита, то есть для соединения подойдут все IP-адреса вида `192.168.0.x`. Кроме того, можно указать значение `samenet` или `samehost`.

Не используйте значение `password`, которое разрешает пароли в виде прос того текста.

`Method = trust` значит «никакой аутентификации». Среди других методов аутентификации назовем GSSAPI, SSPI, LDAP, RADIUS и PAM. Можно также создавать соединения с использованием SSL – тогда аутентификация будет проводиться по SSL-сертификатам клиента.

И еще

В ранних версиях PostgreSQL сетевой доступ разрешался с помощью ключа `-i` в командной строке при запуске сервера. Эта опция доступна и сейчас, но теперь она записывается по-другому:

```
listen_addresses = '*'
```

Так что, если в инструкции по настройке вам встретилось упоминание этой опции, возможно, ваше руководство уже устарело, и не исключено, что многие другие рекомендации тоже не соответствуют действительности.

Смотрите также

Точнее стандартное расположение файлов можно узнать в документации по вашему установщику и/или операционной системе.

Использование графических инструментов администрирования

Системным администраторам часто требуются графические инструменты администрирования.

В PostgreSQL много таких инструментов, но самыми популярными являются следующие:

- pgAdmin3;
- phpPgAdmin.

Первый по умолчанию входит в установщик PostgreSQL в Windows, что, вероятно, делает его самым популярным интерфейсом даже несмотря на то, что чаще на сервере стоит Linux или другая операционная система. Поэтому в книге подробнее рассматривается pgAdmin3.

Как это сделать

pgAdmin3 – это клиентское приложение, которое отправляет и получает SQL с сервера PostgreSQL, отображая для вас результат своей работы. Один клиент pgAdmin может иметь доступ к нескольким серверам PostgreSQL, так же как и один сервер PostgreSQL может принимать несколько клиентов pgAdmin.

Обычно pgAdmin3 сокращают до pgAdmin. Появление тройки в названии – долгая история, но никакого практического смысла здесь нет. Это не номер версии. На момент написания книги самой свежей является версия 1.12.

При первом запуске pgAdmin предложит вам зарегистрировать новый сервер, как показано на рис. 1.1.



Рис. 1.1

Обратите внимание на пять основных параметров соединения.

Предложенный номер порта 2345 не является значением по умолчанию для PostgreSQL (обычно это 5432), что, вероятно, побудит вас выбрать значение, которое подходит именно вам.

Опцию **Store password** следует отключить.

Если у вас несколько серверов баз данных, их можно сгруппировать. Предпочтительней не присваивать серверам *цвета*, поскольку красный, желтый и зеленый обычно используются для индикации статуса. Достаточно придумать для своих серверов разумные имена.

После этого вы получаете доступ к основному окну просмотрщика, в котором слева показано дерево объектов, а справа сверху – их свойства.

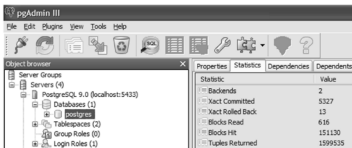


Рис. 1.2

pgAdmin позволяет увидеть множество данных, хранимых в PostgreSQL. Информация подается в контексте, и вы можете быстро и легко найти то, что вам нужно. Динамического обновления здесь нет, поэтому при работе с приложениями не забывайте об *F5*.

В pgAdmin вы будете получать ежедневные подсказки **TIP of the Day** (если подсказки не нужны, можно легко их отключить). А вот подсказки **Guru Hints** лучше держать включенными; к счастью, здесь не встречается никаких жизнерадостных скрепок, постоянно вылезающих на экран со своими советами.

Еще в pgAdmin есть генераторы **Object Report** и **Grant Wizard**, которые хорошо подходят для обзоров и мгновенной поддержки.

Для запросов в pgAdmin создан инструмент **Query**, который позволяет заводить несколько активных сессий. В Query есть замечательная опция **Visual Explain** и **Graphical Query Builder**.

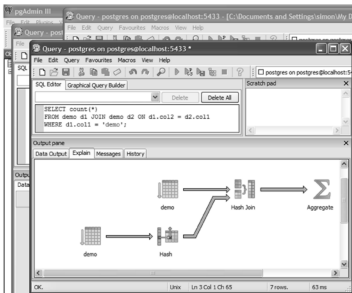


Рис. 1.3

Как это работает

Диапазон возможностей pgAdmin широк, но многое можно делать и с помощью других инструментов. Выберите инструменты, которые вам подходят, ведь для каждого действия понадобится правильный инструмент, и выбор будет зависеть от ваших умений, опыта и личных предпочтений.

pgAdmin передает SQL на сервер PostgreSQL и без лишних сложностей возвращает результат. Просто фантастический просмотрщик! Идеальный для выполнения некоторых (небольших) задач администрирования. И, как можно судить из последней фразы, мы не рекомендуем использовать pgAdmin для выполнения *всех* задач.

Очень важная часть работы администратора баз данных состоит в написании скриптов: вы сохраняете копию задачи, чтобы в случае необходимости отредактировать ее и запустить для повторного выполнения. К тому же можно собрать все задачи в один скрипт и передать его одной транзакцией, чего современные GUI-инструменты пока не

могут. Для этого pgAdmin предлагает pgScript, который работает только с pgAdmin, и это сильно затрудняет портирование. Лучше создавать скрипты с помощью утилиты psql. В ней есть множество дополнительных возможностей, которые вы со временем обязательно оцените.

Утилита psql применяется для написания не только скриптов, но и SQL-запросов. Кому-то это может показаться странным выбором, подходящим только для экспертов, но есть в psql две очень полезные возможности, помогающие в написании запроса – это быстрая справка по SQL и автодополнение. Вы сможете быстро написать запрос, даже если точно не помните синтаксис.

Еще одно короткое замечание. Если при создании объекта в pgAdmin вы используете заглавные буквы, объект создается с именем в смешанном регистре. Если вам нужна таблица MyTable, единственный способ до нее добраться – использовать двойные кавычки: "MyTable". Об этом подробно написано в разделе «Управление объектами с именами, заключенными в кавычки».



Рис. 1.4

И еще

phpPgAdmin можно найти по адресу <http://phppgadmin.sourceforge.net/>.

Демонстрационную версию вы можете запустить прямо в своем браузере, чтобы понять, выполняет ли этот инструмент нужную вам работу. На иллюстрации представлен phpPgAdmin 4, запущенный в Windows Internet Explorer. В настоящее время уже вышла версия 5, и она работает с PostgreSQL 9.



Рис. 1.5

Одно из главных отличий phpPgAdmin от pgAdmin в том, что phpPgAdmin работает в браузере; таким образом, с его помощью задача предоставления безопасного доступа для администраторов может решаться проще.

В phpPgAdmin база представлена в виде традиционного дерева и есть простой инструмент для написания SQL-запросов. Возможно, это именно то, что вам нужно. Многие дополнительные функции pgAdmin здесь будут недоступны, но если часть работы выполнять с помощью программирования, это не будет большой проблемой.

Смотрите также

Возможно, вас заинтересуют коммерческие программные продукты для PostgreSQL. Полный список вы найдете в каталоге программного обеспечения для PostgreSQL по адресу:

<http://www.postgresql.org/download/products/1>.

Общие инструменты для администрирования из этого списка:

- Navicat (<http://pgsql.navicat.com/>);
- EMS SQLManager (<http://www.sqlmanager.net/products/studio/postgresql>);
- Lightning Admin (<http://www.amsoftwaredesign.com/>).

Остальные предназначены для разработки, моделирования данных, шаблонного администрирования.

Использование инструмента для написания запросов и скриптов psql

Утилита для написания запросов psql идет в стандартной поставке PostgreSQL, поэтому доступна и одинаково работает во всех средах, что делает ее идеальным инструментом для разработки портируемых приложений и методов. Утилита psql является как интерактивным инструментом создания запросов, так и средством создания скриптов.

Подготовка

В примерах мы предполагаем, что команды psql достаточно, чтобы получить доступ к вашему серверу PostgreSQL. В данном случае для всех параметров соединения берутся значения по умолчанию, хотя на самом деле это может быть и не так.

Полностью параметры соединения записываются следующим образом:

```
psql -h hostname -p 5432 -d dbname -U username -W
```

Как это сделать

Самая простая команда выполняет одну SQL-команду и выдает результат в виде:

```
$ psql -c "SELECT current_time"
timetz
-----
18:48:32.484+01
(1 row)
```

Ключ `-c` не является интерактивным. Если необходимо выполнить несколько команд, можно записать их в текстовом файле, а потом вызвать его с помощью ключа `-f`. Следующая команда подгружает небольшой и очень простой набор примеров:

```
$ psql -f examples.sql
```

и в случае успеха получаем результат:

```
SET
SET
SET
SET
SET
SET
CREATE SCHEMA
SET
SET
SET
DROP TABLE
CREATE TABLE
DROP TABLE
CREATE TABLE
```

Результат работы скрипта `examples.sql` очень похож на дамп-файл, создаваемый при резервном копировании, то есть такой тип файла и результат является довольно распространенным. Каждый раз при успешном выполнении команды PostgreSQL выдает ее имя в виде тега команды, и получается результат, подобный приведенному выше.

Кроме того, `psql` можно использовать и в интерактивном режиме, который является режимом по умолчанию, поэтому для его включения дополнительные опции не требуются:

```
$ psql
postgres=#
```

Первая интерактивная команда, которая может пригодиться:

```
postgres=# help
```

Далее вы можете вводить SQL-запросы или другие команды.

Последней интерактивной командой должна быть

```
postgres=# \quit
```

К сожалению, нельзя просто написать `quit`, и `\exit` тоже не работает. Только `\quit` или, для краткости, `\q`.

Как это работает

В `psql` можно использовать два типа команд:

- мета-команды `psql`;
- SQL.

Мета-команда предназначена для клиента `psql`, а SQL отправляется серверу БД. Например, `\q` командует клиенту отсоединиться. Предполагается, что любая непустая строка, начинающаяся с символа `\` (обратная косая черта) является мета-командой.

Если это не мета-команда – значит, SQL. SQL продолжается до знака `;` (точка с запятой), поэтому можно при желании записывать его в несколько строк в любом удобном формате.

Единственным исключением является команда `help`: это сделано для новичков и тех, кто запутался в синтаксисе. Справочная информация состоит из двух разделов:

- `\?` – информация по мета-командам `psql`;
- `\h` – информация по командам SQL.

Например:

```
postgres=# \h DELETE
Command:  DELETE
Description: delete rows of a table
Syntax:
DELETE FROM [ ONLY ] table [ [ AS ] alias ]
    [ USING usinglist ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ AS output_name ]
    [,]]
```

Это позволяет узнать и запомнить синтаксис и опции команд.

Стоит также отметить еще одну полезную возможность – перемещение по истории ранее выполненных команд.

Кроме того, поможет автодополнение: вы вводите часть команды, и по нажатию клавиши **Tab** она автоматически дополняется. То же

самое работает и для имен объектов: необходимо ввести несколько первых букв, обеспечивающих уникальность имени, а затем нажать клавишу **Tab**, чтобы получить все, что осталось.

Комментарии, состоящие из одной строки, выделяются двумя дефисами:

```
-- Это комментарий, состоящий из одной строки.
```

Многострочные комментарии выделяются так же, как в языках C и Java:

```
/*  
 * Многострочный комментарий.  
*/
```

Возможно, `psql` пугает новичков своим необычным синтаксисом, но остается надеяться, что вы потратите немного времени, чтобы разобраться в интерфейсе и копнуть немного глубже. `psql` – это одна из самых удивительных частей PostgreSQL и наряду с другими утилитами является невероятно полезным инструментом для выполнения задач администрирования баз данных.

И еще...

Программу `psql` можно запускать с разными версиями PostgreSQL, однако можно получить такое сообщение (рис. 1.6).

Кроме того, под Windows предпочтительней использовать для соединения с сервером эмулятор терминала и запускать `psql` в нем.

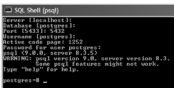


Рис. 1.6

Смотрите также

У `psql` есть и еще масса полезных возможностей:

- информационные функции;
- форматирование вывода;
- запланированное выполнение с помощью команды `\timing`;
- команды ввода/вывода и редактирования;
- файлы автоматического запуска `.psqlrc`;
- заменяемые параметры («переменные»);
- доступ к командной строке ОС.

Безопасная смена пароля

Если вы используете аутентификацию с паролем, этот пароль время от времени следует менять.

Как это сделать

Обычно для этого используют `psql`. Команда `\password` попросит сначала ввести новый пароль, а затем подтвердить его. Соединитесь с `psql` и наберите:

```
\password
```

Введите новый пароль.

При этом `psql` отправит серверу PostgreSQL SQL-выражение, содержащее пароль уже в зашифрованном виде. Например:

```
ALTER USER postgres PASSWORD 'md53175bce1d3201d16594ceb9d7eb3f9d';
```

Ни в коем случае не берите в качестве пароля слово «postgres». Это может сделать вас уязвимым для атак даже самых ленивых хакеров, так что усложните им задачу хоть немного.

И постарайтесь не забыть свой пароль. Трудно управлять базой данных, если невозможно получить к ней доступ.

Как это работает

Так как смена пароля производится одним SQL-выражением, это можно сделать в любом интерфейсе, например:

- pgAdmin3;
- phpPgAdmin.

Если вы не прибегаете ни к одному из перечисленных способов, вы все равно можете изменить пароль, используя SQL в любом интерфейсе. Следует помнить, что необходимо зашифровать пароль, поскольку если вы его отправите в следующем виде:

```
ALTER USER myuser PASSWORD 'secret'
```

он так и попадет на сервер в виде обычного текста, хотя, к счастью, храниться на сервере все же будет в зашифрованном виде.

PostgreSQL не требует регулярной смены пароля, но при этом допускается использовать и более сложные механизмы аутентификации, такие как GSSAPI, SSPI, LDAP, RADIUS и т. п.

Как избежать хранения пароля в коде

Всем понятно, что хранить пароль в явном виде опасно. Ниже рассказывается, как сохранить пароль в защищенном парольном файле.

Подготовка

Не всем пользователям баз данных нужны пароли: для некоторых баз данных используются другие способы аутентификации. Если вы не приверженец аутентификации с паролем, вы можете пропустить этот шаг.

Для начала уберите пароль оттуда, где вы его сохранили. Удалите запись `password=xxxx` из строки соединения программы, иначе при проверке парольного файла пароль, заданный программой, перекроет значение, которое вы бы хотели задать в файле. Оставлять пароль в коде и в парольном файле – тоже не лучшая идея.

Помимо этого, не рекомендуется использовать переменную окружения `PGPASSWORD`, так что уберите и ее.

Если вы считаете, что ваш пароль кто-то видел, смените его, прежде чем сохранить в парольном файле.

Как это сделать

Парольный файл содержит пять полей, которые обычно используются при создании соединения, поэтому можно настроить права доступа к этому файлу, чтобы еще лучше защитить свой пароль:

```
host:port:dbname:user:password
```

то есть

```
myhost:5432:postgres:griggs:moresecure
```

Парольный файл находится с помощью переменной окружения `PGPASSFILE`. Если значение `PGPASSFILE` не задано, то ищется имя файла и его расположение по умолчанию:

- в *nix-системах проверьте `~/.pgpass`;
- в Windows проверьте `%APPDATA%\postgresql\pgpass.conf`, где `%APPDATA%` – поддиректория данных приложения в записи пути (обычно `C:\`).

Не забудьте указать права доступа к этому файлу. Windows не требует настройки прав доступа, хотя файл находится в безопасном месте. В *nix-системах необходимо указать

```
chmod 0600 ~/.pgpass
```


Если вы забудете это сделать, то клиентское приложение PostgreSQL молча проигнорирует файл `.pgpass`. Так что не забудьте!

Как это работает

Многие называют парольный файл `.pgpass`, независимо от того, используют они Windows или нет; не удивляйтесь, если встретите такое название.

В парольном файле может быть много строк. Каждая строка проверяется на соответствие запрашиваемой комбинации `host:port:dbname:user`, пока не будет найдена подходящая, и затем будет использован соответствующий пароль.

Каждый элемент может иметь конкретное значение либо значение * («звездочка»), которая означает, что подходит все. Частичное совпадение не поддерживается. Имея соответствующие права, пользователь потенциально может создать соединение с любой базой данных. Использовать звездочку имеет смысл только в полях `dbname` и `port`. Например:

- `myhost:5432*:sriggs:moresecurepw`
- `myhost:5432:perf:hannu:okpw`
- `myhost*:perf:gabriele:maggioresecurity`

И еще

Все это подходит, если у вас мало серверов БД. Если у вас много отдельных серверов БД, вам лучше воспользоваться сервисным файлом соединений или даже хранить данные для соединения на сервере LDAP.

Использование сервисного файла соединений

Когда вариантов соединения становится слишком много, может пригодиться *сервисный файл соединений*.

Сервисный файл соединений позволяет дать одно имя набору параметров соединения. К нему можно обращаться централизованно, и отдельным пользователям не обязательно знать узел и порт базы данных. Кроме того, он является более стойким к дальнейшим изменениям.

Как это сделать

Сначала создайте файл с именем `pg_service.conf`, содержащий:

```
[dbservice1]  
host=postgres1  
port=5432  
dbname=postgres
```

Можно также скопировать его в `/etc/pg_service.conf` или в другую принятую центральную локацию. Кроме того, допускается задать в переменной `PGSYSCONFDIR` указание на расположение этой директории.

Затем впишите строку соединения в следующем формате:

```
service=dbservice1 user=sriggs
```

Сервис также можно указать с помощью переменной окружения `PGSERVICE`.

Как это работает

Это применимо только к соединениям в `libpq` и не применимо для `JDBC`.

С помощью сервисного файла соединений можно также задавать имя пользователя, хотя это будет означать, что имя пользователя может быть разделяемым.

Файлы `pg_service.conf` и `.pgpass` могут работать одновременно. Допускается использовать только `onp` или что-либо другое – на ваш вкус. Заметим, что файл `pg_service.conf` является разделяемым, поэтому для хранения паролей он не подходит.

Если установить соединение не удается

Этот раздел посвящен тем случаям, когда что-то идет не так.

Помните, что 90 % проблем возникает из-за непонимания, так что немного терпения – и скоро вы вновь окажетесь в строю.

Как это сделать

Если не удастся установить соединение, воспользуйтесь приведенными ниже рекомендациями:

- проверьте точность написания имен БД и пользователя: возможно, вы запрашиваете сервис на одной системе, в то время

как необходимая вам БД находится в другой системе. Проверьте свои учетные данные. Убедитесь, что вы ничего не перепутали и не ввели, например, вместо имени пользователя имя базы данных или наоборот. Если вы получили сообщение `too many connections` (слишком много соединений), возможно, необходимо прервать соединение для другой сессии или подождать, пока администратор не восстановит возможность создавать соединения;

- выявите явные ограничения. Если вы получили сообщение об ошибке

```
pg_hba.conf rejects connection for host...
```

то ваша попытка создать соединение была явно отклонена администратором БД на этом сервере. Вы не сможете создать соединение с текущей клиентской системы с этими учетными данными. Нет особого смысла обращаться к администратору, поскольку вы своими действиями нарушаете явную политику безопасности;

- проверьте неявные ограничения. Если вы получили сообщение об ошибке

```
no pg_hba.conf entry for...
```

то нет явного правила, которое подходило бы для ваших учетных данных. Возможно, это ошибка администратора, что вполне типично для очень сложных сетей. Обратитесь к администратору, чтобы узнать, по какому правилу ваше соединение в следующий раз будет принято (будем надеяться) или явным образом отвергнуто;

- узнайте, работает ли соединение с `psql`. Если вы пытаетесь создать соединение с PostgreSQL с помощью другой утилиты командной строки, переключитесь на `psql`. Если вам удалось создать соединение с `psql`, но не удастся настроить работу основного соединения, то проблема может быть в локальном интерфейсе, который вы используете;
- выясните, запущен ли сервер. Если сервер не запущен, вы не сможете к нему подсоединиться. Иногда проблема связана с определением, к какому серверу вы хотите подключиться. Необходимо ввести узел и порт – проверьте, не перепутали ли вы что-нибудь при вводе;
- узнайте, запущен ли сервер и принимает ли он новые соединения. Сервер, который выключается, не будет принимать

никаких новых соединений – только соединения суперпользователей. Кроме того, соединение не будет создано, если сервер находится в режиме ожидания, у него не включен параметр `hot_standby`;

- проверьте, правильно ли «слушает» сервер;
- проверьте порт, который слушает сервер. Убедитесь в том, что входящий запрос попадает в интерфейс, перечисленный в параметре `listen_addresses`, или что в нем указана * (звездочка) для удаленных соединений или `localhost` для локальных соединений;
- выясните, существуют ли БД и пользователь с указанными именами. Возможно, такая БД или пользователь больше не существуют;
- проверьте запрос на соединение. Возможно, запрос был принят, но затем соединение было сброшено. Вы можете узнать это из журнала сервера, если включены следующие параметры:

```
log_connections = on  
log_disconnections = on
```

- есть и другие причины разрыва соединения. Если вы подключаетесь к серверу, находящемуся в режиме ожидания, возможно, соединение разрывается из-за конфликта режима горячего ожидания.

И еще

С появлением новых версий PostgreSQL области, связанные с аутентификацией клиентов и безопасностью, меняются довольно часто. Изменения можно обнаружить даже в минорных версиях.

Документацию по аутентификации вы найдете по адресу <http://www.postgresql.org/docs/current/interactive/client-authentication.html>.

Прежде чем обратиться к документации или в службу поддержки, не забудьте уточнить, какую версию PostgreSQL вы используете. Путаница в версиях вызывает много проблем.



ГЛАВА 2.

Исследование базы данных

Рассматриваемые темы:

- Как узнать версию сервера
- Определение времени работы сервера
- Где находятся файлы сервера БД
- Где находится журнал сообщений сервера
- Где находится системный идентификатор БД
- Список баз данных на сервере
- Сколько таблиц в базе данных
- Сколько места на диске занимает база данных
- Сколько места на диске занимает таблица
- Какие таблицы являются самыми большими
- Сколько строк в таблице
- Быстрый подсчет строк в таблице
- Понятие объектных зависимостей

Введение

Чтобы понять СУБД PostgreSQL, необходимо увидеть ее в работе. Пустая база данных – это все равно что город-призрак без домов.

Будем считать, что база данных у вас уже есть. Написана тысяча книг о том, как создать базу данных с нуля. В этом же руководстве предлагается помощь тем, у кого есть доступ к базе данных PostgreSQL, но кто до сих пор учится работать с СУБД PostgreSQL.

Лучшее начало – сформулировать ряд простых вопросов, которые помогут вам определить свое положение. Кстати, на эти же вопросы вам понадобится ответить, если вы попросите кого-либо о помощи.

Как узнать версию сервера

При возникновении какой-либо проблемы необходимо проверить и перепроверить, какая у вас версия сервера. Это поможет сообщить об ошибке либо обратиться к нужной версии документации.

Как это сделать

Создайте соединение с базой данных и выполните следующее:

```
postgres # SELECT version();
```

Ответом будет что-то вроде:

```
PostgreSQL 9.0 on x86_64-unknown-linux-gnu,  
compiled by GCC gcc (Ubuntu 4.3.3-5ubuntu4) 4.3.3, 64-bit
```

Наверное, даже слишком много информации для одного раза!

Как это работает

Версии сервера PostgreSQL имеют формат *Мажорная.Минорная.Техническая*.

Для некоторых проектов достаточно знать *Мажорный* номер версии, но в случае с PostgreSQL необходимо знать номер версии на уровне *Мажорная.Минорная*; на этом уровне меняются список возможностей и совместимость. Например, в версии *8.4* больше дополнительных возможностей и изменений в совместимости, чем в *8.3*. Кроме того, для каждой минорной версии есть отдельная версия документации, поэтому если что-то идет не так, необходимо обращаться к *правильной* версии документации.

Техническая часть номера версии дополняет схему. Если *8.4.0* – первоначальный релиз версии *8.4*, то *8.4.1* – это следующий за ним технический релиз.

О политике поддержки релизов вы можете прочитать на странице http://wiki.postgresql.org/wiki/PostgreSQL_Release_Support_Policy.

Там объясняется, что каждый релиз поддерживается в течение пяти лет.

Все релизы до 8.1 включительно перестали поддерживаться в ноябре 2010 года – теперь поддерживаются только версии 8.2 и выше. Ранние версии тоже работают, но в них не хватает многих полезных возможностей и их производительность хуже, чем у более поздних версий. Поддержка последующих версий будет прекращена.

Таблица 2.1.

Версия	Окончание периода поддержки
PostgreSQL 8.2	Декабрь 2011 года
PostgreSQL 8.3	Февраль 2013 года
PostgreSQL 8.4	Июль 2014 года
PostgreSQL 9.0	Август 2015 года (приблизительно)

И еще

Есть и другие способы узнать номер версии:

```
bash # psql --version
psql (PostgreSQL) 9.0
```

Однако имейте в виду, что в данном случае показана версия клиентского программного обеспечения, которая может отличаться от версии сервера. Непосредственно версию сервера можно узнать с помощью

```
bash # cat $PGDATADIRECTORY/PG_VERSION
```

Но ни один из этих способов не покажет технический номер версии.

Почему так важно знать версию СУБД?

В PostgreSQL есть внутренние номера версий для формата файла данных, структуры каталога БД и формата восстановления после сбоя. Все это сервер проверяет во время работы, чтобы убедиться в том, что данные не повреждены. В PostgreSQL эти форматы меняются только со сменой релиза и никогда в рамках одного релиза.

С точки зрения пользователя в каждой версии меняется поведение сервера. Если вы хорошо знаете приложение, которое используете, то поймете разницу, просто прочитав информацию, сопровождающую новый релиз. Но в большинстве случаев безопаснее всего протестировать новую версию в своем приложении.

Определение времени работы сервера

Другими словами, необходимо выяснить, как давно был запущен сервер.

Как это сделать

Выполните следующий SQL-запрос в любом интерфейсе:

```
postgres=# SELECT date_trunc('second',
current_timestamp - pg_postmaster_start_time()) as uptime;
uptime
-----
00:38:15
```

Как это работает

Postgres сохраняет время запуска сервера, которое можно получить следующим образом:

```
postgres=# SELECT pg_postmaster_start_time();
pg_postmaster_start_time
-----
2009-11-26 09:39:23.354208+00
```

Затем можно выполнить SQL-запрос и узнать общее время работы сервера:

```
postgres=# SELECT current_timestamp - pg_postmaster_start_
time();
           ?column?
-----
00:35:30.22868
```

или в другом формате:

```
postgres=# SELECT date_trunc('second',
current_timestamp - pg_postmaster_start_time()) as uptime;
           uptime
-----
00:38:15
```

Смотрите также

Здесь все просто. В последующих главах вы больше узнаете о мониторинге и статистике.

Где находятся файлы сервера БД

Файлы сервера баз данных по умолчанию хранятся в месте под названием *data directory* (директория данных). Дополнительные файлы данных могут также находиться в *tablespaces*.

Подготовка

Вам необходимо будет узнать путь к СУБД в операционной системе или платформе, на которой запущена база данных.

Как это сделать

В *Debian* или *Ubuntu* директория данных по умолчанию

```
/var/lib/postgresql/R.r/main
```

Здесь *R.r* — это мажорная и минорная части номера версии СУБД, например, 9.0. Конфигурационные файлы лежат в следующем месте:

```
/etc/postgresql/R.rNn/main/
```

В обоих случаях *main* — это просто имя сервера БД. Оно может быть другим. Для простоты далее в этой книге будем предполагать, что у вас только один экземпляр БД.

В *Red Hat RHEL*, *CentOS* или *Fedora* путь к директории данных по умолчанию следующий:

```
/var/lib/pgsql/data/
```

Там же по умолчанию находятся конфигурационные файлы (*.conf).

И вновь *data* – это лишь локация по умолчанию. С помощью утилиты *initdb* можно создавать дополнительные директории данных.

В *Windows* и *OS X* путь к директории данных по умолчанию следующий:

```
C:\Program Files\PostgreSQL\R.r\data
```

Как это работает

Хотя файловые структуры *Debian/Ubuntu* и *Red Hat* различаются, они следуют стандарту иерархии файловой системы *Linux (FHS – Filesystem Hierarchy Standard)*, поэтому и та, и другая структура является *правильной*.

Структура *Red Hat* проще и доступнее для понимания. Структура *Debian/Ubuntu* сложнее и используется для решения других, более смелых задач. Структура *Debian/Ubuntu* напоминает гибкую архитектуру (*OFA – Optimal Flexible Architecture*) других СУБД. Требуется создать такую файловую структуру, которая позволит запускать в рамках одной системы несколько серверов БД *PostgreSQL* и будет допускать существование в одной файловой системе нескольких версий программного обеспечения.

Для версий под *Windows* и *OS X* используется другая структура. Она допускает несколько кластеров БД, но все же сложнее, чем *Debian/Ubuntu*.

Предпочтительней все же структура *Debian/Ubuntu*, какую бы платформу вы ни использовали. У нее нет своего имени, поэтому назовем ее гибкой архитектурой *PostgreSQL (PFA – PostgreSQL Flexible Architecture)*. Здесь она не рассматривается детально, поэтому вам придется разложить все по полочкам самостоятельно, но в долгосрочной перспективе это окупится. В *PFA* используются следующие переменные окружения для названия частей файловой структуры:

- `export PGROOT=/var/lib/pgsql/;`
- `export PGRELEASE=9.0;`
- `export PGSERVERNAME=mamba;`
- `export PGDATADIR=$PGROOT/$PGRELEASE/$PGSERVERNAME;`
- например, значение `PGDATADIR` – это `/var/lib/pgsql/9.0/mamba`.

И еще

Как только вы определили директорию данных, вам понадобятся файлы, из которых состоит сервер PostgreSQL.

Таблица 2.2.

Поддиректория	Назначение
base	Основная директория данных. Здесь у каждой БД своя директория, в которой находятся файлы для каждой таблицы БД или индекс
global	Таблицы каталогов сервера БД, разделяемые между всеми БД
pg_clog	Файлы статуса транзакций
pg_multixact	Файлы статуса блокировок на уровне строк
pg_subtrans	Файлы статуса субтранзакций
pg_tblspc	Ссылки на внешние табличные пространства
pg_twophase	«Двухфазный коммит» или статусы подготовленных транзакций
pg_xlog	Журнал транзакций, или журнал опережающей записи (WAL – Write Ahead Log)

Ни в одной из перечисленных директорий нет файлов, которые пользователь мог бы изменить, и не стоит удалять эти файлы вручную. Лучше не трогайте – иначе сломаете, а починить, скорее всего, не получится! Неразумно даже копировать какие-либо файлы в эти директории, не убедившись, что вы тщательно выполнили все инструкции, описанные в десятой главе.

Далее мы еще поговорим о табличных пространствах, а также обсудим улучшение производительности, заключающееся в выделении для журнала транзакций собственных дисков.

Единственное, что можно трогать, – это конфигурационные файлы, то есть файлы *.conf, или файлы журнала сообщений сервера. Они могут находиться в директории данных или в каком-либо другом месте.

Где находится журнал сообщений сервера

Журнал сервера БД – это запись всех сообщений, созданных сервером БД. Именно в него следует смотреть в первую очередь, если

возникают проблемы с сервером, а лучше заглядывать в него регулярно.

Сообщения в нем имеют следующий вид:

```
2010-01-19 21:23:52 GMT LOG: database system was not
properly shut down; automatic recovery in progress
2010-01-19 21:23:52 GMT LOG: record with zero length at
0/49AF90
2010-01-19 21:23:52 GMT LOG: redo is not required
2010-01-19 21:23:52 GMT LOG: autovacuum launcher started
2010-01-19 21:23:52 GMT LOG: database system is ready to
accept connections
```

Обсудим их подробнее, как только выясним, где находятся эти файлы.

Подготовка

Необходимо получить доступ к БД на уровне операционной системы, или, как говорится, платформы, на которой запущена БД.

Как это сделать

Журнал сервера может находиться в разных местах, поэтому для начала выпишем все варианты, а потом выясним, где находится журнал, или решим, куда хотим его поместить:

- журнал сервера может находиться в поддиректории директории данных;
- он может быть в какой-либо директории операционной системы;
- он может перенаправляться в *syslog*;
- журнал сервера вообще может отсутствовать – тогда самое время его добавить.

Если журнал не перенаправляется в *syslog*, он может состоять из одного или нескольких файлов. Имена этих файлов можно менять, поэтому в разных системах они могут быть разными.

В системах Debian или Ubuntu по умолчанию журнал сервера находится в директории

```
/var/log/postgresql
```

Журнал текущего сервера называется `postgresql-N.n-main.log`, где `N.n` – номера мажорного и минорного релизов сервера, например 9.0.

Более старые журналы будут иметь номер `postgresql-8.3-main.log.1`, причем чем больше последнее число, тем старше файл.

В Red Hat, RHEL, CentOS или Fedora журнал сервера находится по умолчанию в поддиректории директории данных:

```
/var/lib/pgsql/data/pg_log
```

В Windows сообщения по умолчанию отправляются в журнал записи событий Windows (eventlog).

Как это работает

Журнал сервера – это просто файл, в который записываются сообщения сервера. Каждое сообщение имеет уровень важности, самый частый из которых `LOG`, хотя есть и другие, перечисленные в таблице.

Таблица 2.3.

Уровень важности PostgreSQL	Значение	Уровень важности Syslog	Журнал Windows
От <code>DEBUG 1</code> до <code>DEBUG 5</code>	Внутренняя диагностика	<code>DEBUG</code>	<code>INFORMATION</code>
<code>INFO</code>	Результат команды для пользователя	<code>INFO</code>	<code>INFORMATION</code>
<code>NOTICE</code>	Полезная информация	<code>NOTICE</code>	<code>INFORMATION</code>
<code>WARNING</code>	Предупреждает о возможных проблемах	<code>NOTICE</code>	<code>WARNING</code>
<code>ERROR</code>	Выполнение команды прервано	<code>WARNING</code>	<code>ERROR</code>
<code>LOG</code>	Для системных администраторов	<code>INFO</code>	<code>INFORMATION</code>
<code>FATAL</code>	Событие, прерывающее только одну сессию	<code>ERR</code>	<code>ERROR</code>
<code>PANIC</code>	Событие, приводящее к остановке сервера	<code>CRIT</code>	<code>ERROR</code>

Обратите внимание на сообщения `FATAL` и `PANIC`. В большинстве случаев при нормальной работе сервера они приходиться не должны, за исключением некоторых случаев, связанных с репликацией (об этом можно прочитать в соответствующей главе).

Число сообщений, появляющихся в журнале, можно задать с помощью параметра сервера `log_min_messages`, а количество информации, показываемой для каждого события, настраивается в параметре `log_error_verbosity`. Если сообщения были отправлены в стандартный

журнал, то в начале каждой строки ставится префикс с полезной информацией, который также можно контролировать с помощью `log_line_prefix`.

Где находится системный идентификатор БД

У каждого сервера БД есть системный идентификатор, присваиваемый при инициализации (создании) БД. Идентификатор сервера сохраняется при резервном копировании, клонировании и т. п.

Многие действия с сервером зависят от системного идентификатора, поэтому вас попросят сообщить эту информацию, если вы сообщите об ошибке.

Как это сделать

Connect as the postgres OS user, or another user with execute privileges on the server software.

```
pg_controldata <data-directory> | grep "system identifier"  
Database system identifier:          5558338346489861223
```

Не используйте `-D` перед названием директории данных. Это возможно только в серверном приложении PostgreSQL.

Как это работает

`pg_controldata` – это серверное приложение PostgreSQL, показывающее содержимое контрольного файла сервера. Контрольный файл находится в директории данных сервера и создается при инициализации БД. Некоторая информация в нем обновляется регулярно или при наступлении определенных важных событий.

Полный вывод `pg_controldata` выглядит так:

Таблица 2.4.

pg_control version number:	851
Catalog version number:	200911051
Database system identifier:	5408338346412861210
Database cluster state:	in production
pg_control last modified:	Thu 26 Nov 2009 09:39:23 GMT
Latest checkpoint location:	0/53D8E0
Prior checkpoint location:	0/53D840

Таблица 2.4.

Latest checkpoint's REDO location:	0/53D8E0
Latest checkpoint's TimeLineID:	1
Latest checkpoint's NextXID:	0/649
Latest checkpoint's NextOID:	11565
Latest checkpoint's NextMultiXactId:	1
Latest checkpoint's NextMultiOffset:	0
Latest checkpoint's oldestXID:	641
Latest checkpoint's oldestXID's DB:	1
Time of latest checkpoint:	Thu 26 Nov 2009 09:15:02 GMT
Minimum recovery ending location:	0/0
Maximum data alignment:	8
Database block size:	8192
Blocks per segment of large relation:	131072
WAL block size:	8192
Bytes per WAL segment:	16777216
Maximum length of identifiers:	64
Maximum columns in an index:	32
Maximum size of a TOAST chunk:	1996
Date/time type storage:	64-bit integers
Float4 argument passing:	by value
Float8 argument passing:	by value

И еще

Ни в коем случае не изменяйте контрольный файл PostgreSQL. В противном случае сервер в следующий раз может не запуститься или могут появиться скрытые ошибки.

Список баз данных на сервере

При создании соединения с PostgreSQL вы всегда подключаетесь к одной БД на одном из серверов. Если на сервере несколько баз данных, может возникнуть путаница. Следовательно, необходимо узнать, какие именно базы данных относятся к нужной СУБД.

Дополнительную путаницу создает тот факт, что у понятия «база данных» есть два различных, хотя и связанных между собой значения. Изначально считалось, что PostgreSQL – это «база данных», в которую пользователь помещает данные. Однако сервер PostgreSQL

может работать с несколькими отдельными базами данных, поэтому следует разделять эти два понятия.

Заметим также, что в некоторых реляционных СУБД, таких как SQLServer, термин «база данных» больше соответствует термину «схема» в PostgreSQL.

Как это сделать

Если у вас есть доступ к `psql`, вы можете набрать следующие команды:

```
bash $ psql -l
List of databases
  Name      | Owner  | Encoding | Collation | Ctype   | Access
privileges
-----+-----+-----+-----+-----+-----
 postgres   | sriggs | UTF8     | en_GB.UTF-8 | en_GB.UTF-8 | 
 template0   | sriggs | UTF8     | en_GB.UTF-8 | en_GB.UTF-8 | =c/sriggs
+
          |      |          |          |          | 
 sriggs=CtC/sriggs
 template1   | sriggs | UTF8     | en_GB.UTF-8 | en_GB.UTF-8 | =c/sriggs
+
          |      |          |          |          | 
 sriggs=CtC/sriggs
(3 rows)
```

Ту же информацию можно получить в `psql`, набрав `\l`.

Вся эта информация хранится в таблице-каталоге PostgreSQL, которая называется `pg_database`. К этой таблице можно обратиться с помощью SQL-запроса и получить следующий результат:

```
postgres=# select datname from pg_database;
 datname 
-----
 template1
 template0
 postgres
(3 rows)
```

Как это работает

При запуске в PostgreSQL есть три базы данных, а именно `template0`, `template1` и `postgres`. `Postgres` является основной базой данных пользователя. Также можно создавать собственные базы данных с помощью команды

```
CREATE DATABASE my_database;
```

или набрав в командной строке

```
bash $ createdb my_database
```

Создав базу данных, убедитесь, что она защищена. Об этом подробно рассказывается в шестой главе.

При создании новой базы данных на самом деле просто копируется одна из существующих БД, после чего эти две базы уже никак не связаны между собой.

Template0 и template1 – это шаблонные базы данных. Template1 можно изменить, чтобы создать локализованный шаблон для всех создаваемых баз данных. Template0 представляет собой резервную копию, к которой вы всегда сможете вернуться, если внесете изменения в template1.

Вы можете удалить базу данных postgres, но давайте договоримся этого не делать. И не трогайте template0. Template1 создана для того, чтобы ее изменять, так что пользуйтесь этой возможностью.

И еще

Информация, рассматриваемая в этом разделе, находится в таблице-каталоге PostgreSQL, которая называется `pg_database`. Можно обращаться непосредственно к этой таблице, но в некоторых случаях результат может оказаться хуже, чем если бы мы искали и в других таблицах:

```
postgres=# \x
postgres=# select * from pg_database;
-[ RECORD 1 ]-+-----
datname      | template1
datdba       | 10
encoding     | 6
datcollate   | en_GB.UTF-8
datctype     | en_GB.UTF-8
datistemplate| t
datallowconn | t
datconnlimit | -1
datlastsysoid| 11620
datfrozensid | 644
dattablespace| 1663
datacl       | ({c/sriggs,sriggs=CTc/sriggs})
-[ RECORD 2 ]-+-----
datname      | template0
datdba       | 10
encoding     | 6
datcollate   | en_GB.UTF-8
datctype     | en_GB.UTF-8
datistemplate| t
datallowconn | f
datconnlimit | -1
```



```

datlastsysoid | 11620
datfrozenxid  | 644
dattablespace | 1663
datacl        | {=c/sriggs,sriggs=CTc/sriggs}
-[ RECORD 3 ]-+-----
datname       | postgres
datdba        | 10
encoding      | 6
datcollate    | en_GB.UTF-8
datctype      | en_GB.UTF-8
datistemplate | f
datallowconn  | t
datconnlimit  | -1
datlastsysoid | 11620
datfrozenxid  | 644
dattablespace | 1663
datacl        |

```

Для начала обратите внимание на использование команды `\x`. Здесь в выводе `psql` каждой строке соответствует один столбец, а не строка. Понятно, что такой формат вызывает много вопросов.

Можно включать и выключать соединение для каждой базы данных и настраивать ограничения для этих соединений.

У каждой базы данных есть табличное пространство по умолчанию. Таким образом, таблицы данных создаются в одной специальной базе данных, а файлы данных для этой таблицы располагаются в одном табличном пространстве.

Также можно отметить, что каждая база данных содержит кользящую последовательность, что позволяет устанавливать различные свойства языка SQL. Подробнее об этом – в следующих разделах.

Сколько таблиц в базе данных

Число таблиц в реляционной базе данных – хороший показатель сложности БД и простой ключ к ее пониманию.

Как это сделать

В любом интерфейсе наберите следующую SQL-команду:

```

SELECT count(*) FROM information_schema.tables
WHERE table_schema NOT IN ('information_schema',
                           'pg_catalog');

```

Это позволит взглянуть на список таблиц и понять, много их или мало.

В `psql` можно увидеть свои таблицы с помощью:

```
postgres@ebony:~/8.3/main$ psql -c "\d"
```

List of relations

Schema	Name	Type	Owner
public	accounts	table	postgres
public	branches	table	postgres

В `pgadmin3` в режиме просмотра в виде дерева таблицы отображаются в правой части экрана, как показано на иллюстрации ниже.

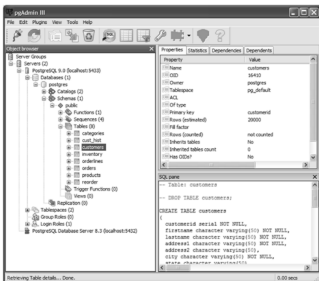


Рис. 2.1

Как это работает

PostgreSQL хранит информацию о базе данных в каталоге таблиц (catalog tables) – там описываются все изменения, внесенные в любую таблицу. Основной каталог таблиц хранится в схеме `pg_catalog`. Есть и другой набор объектов каталога, который называется Information Schema и является стандартным для SQL способом получения информации в реляционных базах данных. Нужно было исключить обе эти схемы из запроса, иначе вы получили бы слишком много

информации. Для этого в часть запроса с `WHERE` вы вставили фразу `NOT IN`.

Заметим, что такой запрос покажет только число строк в одной базе данных на сервере PostgreSQL. Можно видеть таблицы только в той базе данных, с которой установлено соединение, поэтому придется запускать один и тот же запрос для всех баз данных по очереди.

И еще

Как уже было сказано, число таблиц в реляционной базе данных является хорошей мерой сложности этой БД. Сложности чего? Например, нужна база данных, достаточно гибкая, чтобы работать в различных ситуациях, или есть сложный бизнес-процесс, который необходимо отразить в БД с ограниченным числом объектов. Таким образом, большое количество страниц может свидетельствовать о сложном бизнес-процессе или быть просто частью сложного программного обеспечения.

Авторы книги видели базу данных с 20 000 отдельными таблицами, не считая сегментов, представлений или рабочих таблиц. Очевидно, это была очень сложная система.

Таблица 2.5.

Число отдельных таблиц («сущностей»)	Уровень сложности
20 000	Очень сложная. Или вы ошиблись в подсчетах, или вам понадобится большая команда, чтобы справиться с этой базой данных
2 000	База данных сложного проекта; такое встретишь не часто
200	Обычная рабочая база данных
2	База данных, предназначенная для решения одной определенной задачи
0	Вы еще не успели подгрузить данные...

Вы, конечно, не всегда можете с легкостью определить, какие таблицы являются сущностями, поэтому достаточно просто сосчитать таблицы. В некоторых базах данных используется множество сегментов или похожих таблиц, и их количество начинает зашкаливать. Встречаются базы данных с 200 000 таблиц (всех типов). Но доводить до этого дело крайне нежелательно, поскольку каталоги таблиц базы данных разрастаются до невозможности.

Сколько места на диске занимает база данных

Необходимо знать размер таблицы, чтобы грамотно использовать дисковое пространство.

Как это сделать

Есть два способа:

- определить размер файлов, из которых состоит сервер БД;
- выполнить SQL-запрос, позволяющий узнать размер базы данных.

Если брать размер реальных файлов, нужно включить директорию данных и все ее поддиректории, а также все директории, содержащие табличные пространства. Не самая легкая задача, к тому же трудно собрать все и не пропустить ни одной части.

Проще обратиться к базе данных с таким запросом:

```
SELECT pg_database_size (current_database());
```

Но так вы ограничиваетесь только текущей базой данных. Если нужно узнать суммарный размер всех баз данных, запрос будет таким:

```
SELECT sum(pg_database_size (datname)) from pg_database;
```

Как это работает

Серверу БД известно, какие таблицы подгружены, а кроме того, известно, как вычислить объем каждой таблицы, так что функция `pg_database_size()` просто определяет размеры файлов.

Сколько места на диске занимает таблица

Каков размер таблицы? Как собрать все части таблицы?

Как это сделать

Определить размер таблицы помогает следующий запрос:

```
postgres=# select pg_relation_size('accounts');
pg_relation_size
-----
0
(1 row)
```

Можно узнать также полный размер таблицы, включая индексы и другие связанные с ней пространства, с помощью такого запроса:

```
postgres=# select pg_total_relation_size('accounts');
pg_total_relation_size
-----
0
(1 row)
```

А также можно использовать команду `psql`:

```
postgres=# \dt+ accounts
List of relations Schema | Name
+-----+-----+-----+-----+-----+-----+
| Type | Owner | Size | Description | public |
+-----+-----+-----+-----+-----+
pgbench_accounts | table | 13 MB | | public |
(1 row)
```

Как это работает

Таблицы в PostgreSQL состоят из множества отношений. Главное отношение – это таблица данных. Кроме того, есть всевозможные дополнительные файлы данных. Каждый индекс, создаваемый к таблице, также является отношением. Длинные значения располагаются во вторичной таблице, которая называется TOAST, поэтому в большинстве случаев у каждой таблицы есть еще таблица TOAST и индекс TOAST.

Каждое отношение состоит из нескольких файлов данных. Основные файлы данных разбиваются на части по 1 ГБ. У первого файла нет никакого суффикса, остальным приписывается номер (например, .2). Кроме того, есть файлы `.vm` и `.fsm`, – это служебные файлы Visibility Map и FreeSpaceMap, и даже для очень больших таблиц они сравнительно небольшие.

И еще

TOAST – это The Outsize Attribute Storage Technique (метод хранения слишком больших атрибутов). Из названия понятно, что это механизм для хранения длинных значений столбцов. В PostgreSQL допускаются значения любых типов данных, размер которых не превышает 1 ГБ. Большие значения разбиваются на несколько частей, поэтому один и тот же тип данных может использоваться для значений от 1 байта до 1 гигабайта.

Какие таблицы являются самыми большими

Как определить размер одной таблицы, вы уже знаете. Посмотрим теперь на эту задачу с другой стороны: выясним не абсолютный, а относительный размер таблиц.

Как это сделать

Десятку «лидеров» показывает такой запрос:

```
SELECT      table_name
            ,pg_relation_size(table_name) as size
FROM information_schema.tables
WHERE table_schema NOT IN ('information_schema',
                           'pg_catalog')
ORDER BY size DESC
LIMIT 10;
```

Список таблиц, ограниченный десятью строками, выводится в порядке убывания размера. В данном случае рассматривались все таблицы во всех схемах, кроме таблиц в `information_schema` или `pg_catalog`. В этих схемах Postgres хранит свои внутренние данные.

Сколько строк в таблице

Подсчет – это самая простая операция в SQL, поэтому многие новички в Postgres начинают именно с него.

Как это сделать

```
SELECT count(*) FROM table;
```

Результатом будет одно целое число. Так это выглядит в `psql`:

```
postgres=# select count(*) from orders;
 count
-----
    345
(1 row)
```

Как это работает

SQL-функция `count(*)` проверяет все строки таблицы методом *последовательного сканирования* (sequential scan): по очереди обращается к блокам данных таблицы и считывает число строк в каждом блоке.

Если таблица находится на диске, то этот метод – выгодная модель получения доступа к диску, быстро возвращающая результат.

Некоторые думают, что такой подсчет является хорошим показателем производительности системы управления базами данных. В отдельных СУБД даже специально оптимизируют выполнение SQL-функции `count`, однако в Postgres этого нет. Среди разработчиков PostgreSQL уже не раз поднимался вопрос о добавлении этой возможности, но мало кто видит необходимость в такой оптимизации. Да, функция `count` часто используется в приложениях, но подсчет без дополнения `where` не столь уж полезен. Так что в настоящее время нет возможности оптимизировать подобный SQL-запрос.

Причиной сканирования таблицы по частям является основное свойство PostgreSQL – *MVCC, контроль совместимости версий* (Multi-Version Concurrency Control). MVCC позволяет вести подсчет одновременно со вставкой, обновлением или удалением данных из таблицы. Это очень полезно, хотя и требует больших усилий от сообщества разработчиков СУБД PostgreSQL.

MVCC позволяет добавлять информацию в каждую строку таблицы с указанием, когда было сделано это изменение. Если изменения были внесены после начала выполнения SQL-функции, то при подсчете они просто игнорируются. Таким образом, чтобы получить результат функции `count`, необходимо для каждой строки таблицы проводить «проверку видимости».

Точнее говоря, результатом подсчета будет только значение в определенный момент времени. В зависимости от того, что происходит с таблицей, это значение может сильно или незначительно измениться в процессе подсчета. Таким образом, получив результат, следует иметь в виду: он означает только то, что *некоторое время назад* в таблице было X строк.

Быстрый подсчет строк в таблице

Не всегда нужно знать точное число строк в таблице; особенно это касается больших таблиц, для которых подсчет может занять слишком много времени. Однако для того, чтобы оценить время выполнения различных операций, администраторам требуется хотя бы приблизительно определить размер таблицы.

Как это сделать

Быстро определить, сколько строк в таблице, можно способом подсчета, который использует оптимизатор Postgres:

```
SELECT (CASE WHEN reltuples > 0 THEN
    pg_relation_size('mytable') / (8192 * relpages /
    reltuples)
    ELSE 0
    END)::bigint AS estimated_row_count
FROM pg_class
WHERE oid = 'mytable'::regclass;
```

И получаем:

```
estimated_count
-----
293
(1 row)
```

Такой результат выдается очень быстро, независимо от размеров оцениваемой таблицы.

Как это работает

Вам уже встречалась функция `pg_relation_size()`, и вы уже знаете, что она возвращает точное значение текущего размера таблицы.

При выполнении запроса `vacuum` в Postgres вы записываете в каталог `pg_class` по два значения для каждой таблицы: число блоков данных (`relpages`) и число строк в таблице (`reltuples`). Некоторые считают, что достаточно для оценки использовать значение `relpages` в `pg_class`, однако на деле оно может оказаться устаревшим. Также нельзя полагаться на информацию из таблицы `pg_stat_user_tables`, подробнее о которой рассказывается в десятой главе.

Оптимизатор Postgres использует `relpages` и `reltuples` для подсчета среднего числа строк в блоке или средней плотности кортежа.

Если предположить, что средняя плотность кортежа с течением времени меняться не будет, то можно подсчитать число строк по формуле: *число блоков данных × число строк в блоке*.

Кроме того, в коде учтены возможные варианты, когда значения `reltuples` или `relpages` равны 0. Вообще-то оптимизатор Postgres работает в этом случае немного лучше, так что полученная оценка довольно приблизительно.

И еще

Этот рецепт хорош тем, что время его выполнения не зависит от размера таблицы. Но плохо то, что для работы функции `pg_relation_size()`

требуется блокировка таблицы, и если у другого пользователя для этой таблицы есть `AccessExclusiveLock`, то придется подождать, пока блокировка будет снята, – только тогда вы сможете узнать результат.

Так что же такое `AccessExclusiveLock`? При выполнении какого-либо действия, например изменения типа данных столбца, PostgreSQL блокирует все остальные действия с этой таблицей. Администратор сталкивается с этим, когда производит какие-либо действия типа `ALTER TABLE`. Он обнаруживает, что запрос выполняется гораздо дольше, чем ожидалось. В такой момент администратора охватывает паника: «Неужели таблица больше, чем я думал? Сколько же мне придется ждать?» Да, лучше было бы узнать об этом заранее, но такие запоздалые мысли уже не спасут положение. Так что нужен способ для оценки размера таблицы без необходимости ее блокировать.

Наиболее верное решение – обратиться к системным файлам, которые PostgreSQL использует для хранения данных, и оценить их размеры.

Тут возникает другая трудность: если таблица заблокирована, то над ней производятся какие-либо операции. Таким образом, обращение к системным файлам может быть либо безрезультатным, либо дающим неверный результат. Тем не менее можно предпринять следующие шаги.

Во-первых, некоторую информацию можно взять из `pg_class`:

```
SELECT reltablespace, relfilenode FROM pg_class
WHERE oid = 'orders'::regclass;
```

Во-вторых, следует уточнить `databaseid` базы данных, к которой принадлежит таблица:

```
SELECT oid as databaseid FROM pg_database
WHERE datname = current_database();
```

Вместе `reltablespace`, `databaseid` и `relfilenode` помогут определить нужные файлы в директории данных.

Если `tablespaceid` равно 0, то файлы будут в следующей директории:

```
$PGDATADIR/base/{databaseid}/{relfilenode}*
```

Чем больше таблица, тем больше файлов вы увидите.

Если `reltablespace` имеет ненулевое значение, то файлы будут в директории

```
$PGDATADIR/pg_tblspc/{reltablespace}/
{databaseid}/{relfilenode}*
```

Размер каждого файла (кроме последнего) должен быть 1 ГБ.

Кроме того, следует кое-что уточнить:

- в PostgreSQL термины «блок данных» и «страница» означают одно и то же, и одинаковый смысл имеют термины «кортеж» и «строка»;
- размер блока данных по умолчанию 8192 байта, однако вы можете задать свое значение, если перекомпилируете сервер и создадите новую базу данных.

Функция 1

Для выполнения подсчета можно создать SQL-функцию, чтобы каждый раз не печатать все команды вручную.

```
CREATE OR REPLACE FUNCTION estimated_row_count(text)
RETURNS bigint
LANGUAGE sql
AS $$
SELECT (CASE WHEN reltuples > 0 THEN
          pg_relation_size($1)/(8192*relpages/reltuples)
        ELSE 0
        END)::bigint
FROM pg_class
WHERE oid = $1::regclass;
$;
```

Функция 2

```
CREATE OR REPLACE FUNCTION
  pg_relation_size_nolock(tablename regclass)
RETURNS BIGINT
LANGUAGE plpgsql
AS $$
DECLARE
  classoutput      RECORD;
  tsid              INTEGER;
  rid               INTEGER;
  dbid              INTEGER;
  filepath          TEXT;
  filename          TEXT;
  datadir           TEXT;
  i                 INTEGER := 0;
  tablesize         BIGINT;
BEGIN
  --
  -- get data directory
  --
  EXECUTE 'SHOW data_directory' INTO datadir;
  --
  -- get relfilenode and reltablespace
```

```
--
SELECT
    reltablespace as tsid
,relfilenode as rid
    INTO classoutput
FROM pg_class
WHERE oid = tablename
AND relkind = 'r';
--
-- Throw an error if we can't find the tablename
specified
--
IF NOT FOUND THEN
    RAISE EXCEPTION 'tablename % not found', tablename;
END IF;
tsid := classoutput.tsid;
rid := classoutput.rid;
--
-- get the database object identifier (oid)
--
SELECT oid INTO dbid
FROM pg_database
WHERE datname = current_database();
--
-- Use some internals knowledge to set the filepath
--
IF tsid = 0 THEN
    filepath := datadir || '/base/' || dbid || '/' || rid;
ELSE
    filepath := datadir || '/pg_tblspc/'
|| tsid || '/'
|| dbid || '/'
|| rid;
END IF;
--
-- Look for the first file. Report if missing
--
SELECT (pg_stat_file(filepath)).size
INTO tablesize;
--
-- Sum the sizes of additional files, if any
--
WHILE FOUND LOOP
    i := i + 1;
    filename := filepath || '.' || i;
    --
    -- pg_stat_file returns ERROR if it cannot see file
    -- so we must trap the error and exit loop
    --
BEGIN
```

```
SELECT tablesize + (pg_stat_file(filename)).size
INTO tablesize;
EXCEPTION
WHEN OTHERS THEN
EXIT;
END;
END LOOP;
RETURN tablesize;
END;
$;$
```

Код будет работать и в Windows, однако потребуется внести некоторые изменения. Попробуйте выполнить это в качестве самостоятельного упражнения.

Понятие объектных зависимостей

В большинстве СУБД существуют зависимости между объектами баз данных. Иногда для выполнения определенных действий необходимо понимать эти зависимости. Рассмотрим их подробнее.

Подготовка

В качестве примера будем использовать простую базу данных, состоящую из двух таблиц:

```
CREATE TABLE orders (
  orderid integer PRIMARY KEY
);
CREATE TABLE orderlines (
  orderid integer
, lineid smallint
, PRIMARY KEY (orderid, lineid)
);
```

Добавим между ними связь, чтобы получилось то, что называется «ссылочная целостность» (referential integrity):

```
ALTER TABLE orderlines ADD FOREIGN KEY (orderid)
REFERENCES orders (orderid);
```

Теперь, если попробовать сделать DROP «ссылочной» таблицы, сервер выдаст сообщение:

```
DROP TABLE orders;
ERROR: cannot drop table orders because other objects
depend on it
DETAIL: constraint orderlines_orderid_fkey on table
```

```
orderlines depends
on table orders
HINT: Use DROP ... CASCADE to drop the dependent objects too.
```

Будьте осторожны! Последовав совету, вы можете случайно удалить все объекты, зависящие от таблицы `orders`. Кому-то это может показаться удобным, однако такие действия – признак лени и недалекости. Возможно, это и сработает, но сначала нужно убедиться, что это сработает как надо.

Итак, необходимо узнать, какими зависимостями связана таблица `orders`, и изучить их. А потом можно будет принять решение, стоит ли использовать `CASCADE` или целесообразнее все сделать вручную.

Как это сделать

Чтобы получить полную информацию о таблице, наложенных на нее ограничениях и ссылках, можно выполнить в `psql` следующую команду:

```
\d+ orders
```

Подробнее об ограничениях можно узнать с помощью запроса:

```
SELECT * FROM pg_constraint
WHERE confrelid = 'orders'::regclass;
```

К сожалению, это еще не все, так что не забудьте прочитать раздел «И еще».

Как это работает

Когда вы создаете внешний ключ (`foreign key`), в каталог `pg_constraint` добавляется ограничение. Запрос показывает, как обнаружить все ограничения, связанные с таблицей `orders`.

Запись

```
WHERE confrelid = 'orders'::regclass;
```

представляет типы идентификаторов объектов. Это упрощенный переход от имени объекта к его числу идентификатора. Проще всего понять это, если иметь в виду, что подобная запись означает то же самое, что и функция `relname2relid()`.

И еще

Когда вы имеете дело с PostgreSQL, всегда можно обнаружить немного больше, если копнуть немного глубже. В данном же случае обнаруживается *гораздо* больше, и это важно.

Приведенные выше запросы показывают только зависимости между таблицами. Еще не обсуждались связи с другими типами объектов. Тем не менее возможны связи с двумя важными типами объектов – представлениями и функциями.

Если ввести команду

```
DROP TABLE orders;
```

связь с любым представлением не даст ей выполниться. Таким образом, сначала следует удалить эти представления и лишь затем таблицу.

С функциями все не так просто. Отношения между функциями и таблицами не записываются в каталог, как и отношения между функциями. Это частично вызвано тем, что многие функциональные языки допускают выполнение динамических запросов, поэтому заранее не известно, какие таблицы или функции будут затронуты при работе функции. Некоторые функции явно ссылаются на таблицы и другие функции, и можно идентифицировать и записать такие зависимости. Однако прямо сейчас мы этого делать не будем. Так что запомните, что информацию о связях функций следует записывать вручную, и только тогда вы точно будете знать, можно ли удалять или переименовывать таблицу либо другие объекты, которые связаны с функциями.



ГЛАВА 3.

Конфигурация

Рассматриваемые темы:

- «Правильная» документация (RTFM)
- Планирование новой базы данных
- Изменение параметров в программах
- Текущие конфигурационные настройки
- Параметры, не определяемые по умолчанию
- Обновление файла параметров
- Настройка параметров для определенных групп пользователей
- Основной список параметров для проверки конфигурации сервера
- Добавление внешних модулей в PostgreSQL
- Запуск сервера в энергосберегающем режиме

Введение

Авторов книги часто спрашивают о параметрах в PostgreSQL. Все куда-то спешат и хотят получить быструю консультацию, чтобы за пять минут выяснить, как все работает. Что ж, попробуем объяснить.

Некоторые администраторы полагают, что можно найти такие волшебные параметры, которые существенно улучшат производительность сервера, и часами роются в книгах, чтобы составить эту чудесную комбинацию. Другие находят какой-нибудь сайт в Интернете, который «объясняет все», и с тех пор убеждены, что их сервер работает как надо.

По большей части ничего сложного тут нет, но идеальные настройки найти трудно. В некоторых случаях оптимальные настройки с течением времени меняются. В данном разделе объясняется, как, когда и где изменять значения параметров.

«Правильная» документация (RTFM)

Обычно аббревиатуру RTFM расшифровывают как «оставьте меня в покое, мне не до вас» или даже используют как оскорбление. Но,

как ни странно, отсылая вас к документации, вам дают хороший совет! Не обижайтесь, примите его. Не забудьте только, что версия документации должна соответствовать версии сервера, которую вы используете.

Документация по PostgreSQL написана очень хорошо и достаточно полно раскрывает многие сложные задачи. Проблема в том, что ее структура не рассчитана на человека, который только начинает свое знакомство с PostgreSQL. Статьи скомпонованы для удобства тех, кто ищет ответы на определенные технические вопросы, чтобы можно было понять, вызваны ли возникшие проблемы ошибкой пользователя. Иногда документация отвечает на вопрос «Что?», но редко на вопросы «Как?» или «Почему?».

Авторы книги тоже принимали участие в написании некоторых разделов документации, однако не предлагают вам обратиться именно к этим пунктам. В RTFM вообще много полезного.

Как это сделать

Основная документация для всех релизов находится по адресу <http://www.postgresql.org/docs/manuals/>.

Среди самых популярных разделов можно назвать следующие:

- SQL Command Reference, Client, and Server tools reference – <http://www.postgresql.org/docs/9.0/interactive/reference.html>;
- Configuration – <http://www.postgresql.org/docs/9.0/interactive/runtime-config.html>;
- Functions – <http://www.postgresql.org/docs/9.0/interactive/functions.html>.

Кроме того, на сайте выложена PDF-версия, в которой иногда проще найти нужную информацию. Только не нужно ее распечатывать! Вся документация занимает примерно 2000 страниц формата A4.

Как это работает

Документация по PostgreSQL написана на языке SGML, который напоминает XML, хотя и отличается от него. Готовые файлы можно конвертировать в HTML, PDF и другие форматы.

И еще

Небесполезно будет заглянуть в wiki-справочник по адресу <http://wiki.postgresql.org>.

Планирование новой базы данных

Планирование новой базы данных может показаться пугающей задачей. Довольно легко запутаться, так что лучше воспользуйтесь идеями, которые представлены в этом разделе. А чтобы еще более упростить задачу, считайте, что при планировании следует принимать во внимание только то, что вы знаете.

Подготовка

Вы уже готовы. Не ждите указаний, что делать. Если вам не озвучили никаких требований, сядьте и напишите их самостоятельно, имея, однако, в виду, что это все-таки скорее предположения, чем требования. Не стоит смешивать эти два понятия.

Повторяйте, пока не придете к какому-либо решению, после чего можно начинать строить прототип.

Как это сделать

Создайте документ, состоящий из следующих пунктов:

1. Структура базы данных. Запланируйте структуру будущей базы данных.
Вычислите первоначальный размер БД.
2. Анализ транзакций:
 - определите, каким образом будет получен доступ к базе данных;
 - посмотрите самые частые пути доступа;
 - узнайте требования к времени отклика.
3. Конфигурация оборудования.
Подумайте о производительности – уместятся ли ваши данные в оперативной памяти?
4. План размещения.
Определите кодировку сервера, локаль и часовой пояс.
5. Доступ к базе данных и безопасность:
 - задайте систему клиентов и необходимые драйверы;
 - создайте роли согласно запланированному контролю доступа;
 - определите `pg_hba.conf`.
6. План поддержки. Кто будет обслуживать БД? Каким образом?

7. Планирование доступности:
 - рассмотрите требования к доступности;
 - `checkpoint_timeout`;
 - запланируйте и протестируйте механизм резервного копирования.
8. Планирование расширенной доступности.
Выберите форму репликации, если она вам вообще нужна.

Как это работает

Одна из главных причин столь раннего планирования заключается в том, что в готовую базу данных некоторые вещи внести очень не просто, если вообще возможно. В особенности это касается кодировки и локали, изменение которых потребует остановки и длительного перерыва в работе. Да и настройки безопасности «на лету» изменить гораздо труднее.

И еще

Планирование полезно всегда. Возможно, вы знаете, что делаете, но другие-то не в курсе! Сообщите другим, что вы собираетесь сделать, прежде чем приступите к работе, – и сэкономите много времени. Не уверены – постройте прототип, который поможет вам принять решение, и выполняйте задачу администрирования как разработчик. Выпишите вещи, которых вы пока не знаете, и постепенно проработайте их.

Авторы книги сознательно опускают многие подробности. У каждого свои методы работы, и очень важно не навязывать свое решение там, где этого не требуется. Если у вас уже есть план – отлично. Если нет – подумайте о том, что вам необходимо, напишите список нужных дел, а затем выполните их.

Изменение параметров в программах

PostgreSQL позволяет настраивать параметры для каждой сессии или для каждой транзакции.

Как это сделать

Значение можно изменить прямо во время сессии:

```
SET work_mem = '16MB' ;
```

Оно будет использоваться для всех последующих транзакций. Можно задать значение только для текущей транзакции:

```
SET LOCAL work_mem = '16MB' ;
```

Это значение будет действовать до конца транзакции или пока вы не выполните команду

```
RESET work_mem;
```

или

```
RESET ALL;
```

SQL-команды SET и RESET можно вызвать в любом интерфейсе. Они применяются только к параметрам сервера PostgreSQL, то есть к таким параметрам, которые влияют на сервер, хотя, возможно, не на весь. Есть и другие параметры, например параметры драйвера JDBC, которые подобным способом изменить нельзя. О них еще пойдет речь ниже.

Как это работает

Если вы изменяете значение параметра во время сессии

```
SET work_mem = '16MB' ;
```

в каталоге pg_settings это будет отображаться следующим образом:

```
postgres=# SELECT name, setting, reset_val, source
           FROM pg_settings WHERE source = 'session';
   name   | setting | reset_val | source
-----+-----+-----+-----
work_mem | 16384   | 1024      | session
```

до тех пор, пока вы не вызовете команду

```
RESET work_mem;
```

после чего значение этого параметра вернется к reset_val, а значение source – к значению по умолчанию.

```
   name   | setting | reset_val | source
-----+-----+-----+-----
work_mem | 1024    | 1024      | default
```

И еще

Значение параметра можно изменить и во время транзакции:

```
SET LOCAL work_mem = '16MB' ;
```

В каталоге pg_settings это отобразится так:

```
postgres=# SELECT name, setting, reset_val
           FROM pg_settings WHERE source = 'session';
   name   | setting | reset_val | source
-----+-----+-----+-----
work_mem | 1024    | 1024      | session
```

Что произошло с параметром? SET LOCAL имеет силу только в той транзакции, во время которой была выполнена в данном случае команда SET LOCAL. Необходимо выполнить ее внутри блока транзакций, и тогда вы сможете увидеть, как изменяется эта настройка:

```
BEGIN;  
SET LOCAL work_mem = '16MB' ;
```

что отразится в представлении каталога pg_settings следующим образом:

```
postgres=# SELECT name, setting, reset_val, source  
          FROM pg_settings WHERE source = 'session';  
   name   | setting | reset_val | source  
-----+-----+-----+-----  
work_mem | 16384   | 1024      | session
```

Следует также заметить, что значением source является session, а не transaction, как можно было бы ожидать.

Текущие конфигурационные настройки

В какой-то момент у вас может возникнуть вопрос: «Каковы же текущие конфигурационные настройки?» Настало время это выяснить.

Как это сделать

Сразу приходит мысль заглянуть в файл postgresql.conf. И это работает, но только если у вас один файл с параметрами. Если файлов два, возможно, вы откроете не тот. Поэтому лучше доверять не текстовому файлу, а самому серверу.

Из предыдущего рецепта вы узнали, что для каждого параметра есть определенная область, где его значение можно изменять. Значения некоторых параметров можно задавать в файле postgresql.conf, зато другие можно определять позже. Таким образом, текущие конфигурационные параметры можно последовательно изменить.

Вы можете воспользоваться командой SHOW:

```
postgres=# SHOW work_mem;  
work_mem  
-----  
1MB  
(1 row)
```

не забывая при этом, что вы получите значения параметров, действительные только на момент ее запуска.

Кроме того, текущие параметры можно посмотреть в виде каталога `pg_settings`.

```
postgres=# \x
Expanded display is on.
postgres=# SELECT * FROM pg_settings WHERE name = 'work_mem';
[ RECORD 1 ] -----
-----
name       | work_mem
setting    | 1024
unit       | kB
category   | Resource Usage / Memory
short_desc | Sets the maximum memory to be used for query
workspaces.
extra_desc | This much memory can be used by each internal
sort operation and hash table before switching to temporary
disk files.
context    | user
vartype     | integer
source     | default
min_val    | 64
max_val    | 2147483647
enumvals   |
boot_val   | 1024
reset_val  | 1024
sourcefile |
sourceline |
```

Таким образом, вы можете узнать значение параметра с помощью команды `SHOW` или получить полную детализацию, обратившись к каталожной таблице.

Как это работает

Каждое значение параметра кэшируется на время сессии, так что вы всегда сможете быстро получить доступ к значениям параметров.

Следует помнить, что полученные значения не обязательно действуют для всего сервера – многие из них определены только для текущей сессии.

Параметры, не определяемые по умолчанию

Теперь попробуем узнать, какие параметры были изменены и правильно ли действуют эти изменения.

Как это сделать

```
postgres=# SELECT name, source, setting
FROM pg_settings
WHERE source != 'default'
      AND source != 'override'
ORDER by 2, 1;
-----name-----|-----source-----|..setting
-----+-----+-----
application_name    | client              | postgres
log_timezone        | command line        | GB
TimeZone            | command line        | GB
timezone_abbreviations | command line        | Default
archive_command     | configuration file   | 
(disabled)
archive_mode        | configuration file   | off
archive_timeout     | configuration file   | 5
bgwriter_delay      | configuration file   | 10
checkpoint_timeout  | configuration file   | 30
log_checkpoints     | configuration file   | on
log_destination     | configuration file   | stderr
log_filename        | configuration file   | log%Y
logging_collector    | configuration file   | on
log_line_prefix     | configuration file   | %t[%p]
log_min_messages    | configuration file   | log
max_prepared_transactions | configuration file   | 5
max_standby_delay   | configuration file   | 90
port                | configuration file   | 5443
max_stack_depth     | environment variable | 2048
work_mem            | session              | 204800
(29 rows)

('Override' is excluded just for display purposes.)
```

Как это работает

В `pg_settings` можно увидеть, значения каких параметров не являются значениями по умолчанию и откуда взялись текущие значения.

Выполнив команду `SHOW`, вы не узнаете, какие значения были изменены, — она просто сообщит все текущие значения. Однако выяснить, какие именно настройки были изменены и почему, вам не удастся.

Если источником является конфигурационный файл, то также указаны значения столбцов `sourcefile` и `sourceline`. Это помогает понять, откуда взялась такая конфигурация.

И еще

Столбец `setting` в `pg_settings` показывает текущие значения, в столбце `boot_val` отображаются значения, которые были определены при

инициализации кластера БД PostgreSQL (`initdb`), и значения, которые будут присвоены после выполнения команды `RESET - reset_val`.

Кто это сделал?

Значение `max_stack_depth` является исключением: хотя в `pg_settings` сказано, что оно было задано через переменную окружения, на самом деле в системах Linux/Unix оно определяется с помощью `ulimit -s`. Значение `max_stack_depth` нужно задавать непосредственно только в Windows.

Значение для часового пояса также берется из окружения ОС, поэтому нет необходимости указывать его отдельно. В `pg_settings` в качестве источника этого значения указывается командная строка.

Обновление файла параметров

Файл параметров является основным местом, где определяются значения параметров сервера PostgreSQL. Все параметры могут быть заданы в файле `postgresql.conf`.

Кроме него есть еще два файла — `pg_hba.conf` и `pg_ident.conf`. Оба имеют отношение к соединениям и безопасности и будут рассмотрены в соответствующих разделах.

Подготовка

Сначала найдите файл `postgresql.conf`, как было описано выше.

Как это сделать

Все параметры можно задать в файле `postgresql.conf`. Некоторые из них имеют влияние только при первом запуске сервера. Типичным примером такого параметра является `shared_buffers`, который определяет объем кэша разделяемой памяти.

Многие параметры можно изменить при работающем сервере. После внесения необходимых изменений на сервере запускается операция `reload`, которая заставляет сервер PostgreSQL прочесть заново файл `postgresql.conf`.

```
pg_ctl -D data reload
```

`postgresql.conf` — это обычный текстовый файл, который можно легко отредактировать. В этом файле перечислено большинство параметров, поэтому вам достаточно будет найти нужный и просто переписать его значение.

Как это работает

Если один и тот же параметр упоминается в файле дважды, сервер будет использовать последнее значение. Поэтому, если вы каждый раз будете дописывать значение в конец файла, может возникнуть путаница, так что лучше этого не делать.

Не добавляйте в файл строки, а только меняйте готовые, или создайте пустой файл, в который будете вносить только те параметры, значения которых нужно изменить. Авторы книги предпочитают работать с файлом, в который собраны только значения, отличающиеся от значений по умолчанию. Так проще следить за происходящим.

Но какой бы метод вы ни выбрали, обязательно сохраняйте все предыдущие версии файлов `.conf`. Их можно просто копировать или использовать систему контроля версий, например SVN.

И еще

В файле `postgresql.conf` поддерживается директива `include` (включение). То есть вы можете вставлять в него ссылки на другие файлы, а в них – ссылки на третьи файлы и т. д. Это позволит лучше организовать настройки, если, конечно, не слишком все усложнять.

Настройка параметров для определенных групп пользователей

В PostgreSQL поддерживается множество способов настройки параметров для различных групп пользователей.

Как это сделать

Для всех пользователей в БД `saas`:

```
ALTER DATABASE saas
SET configuration_parameter = value1;
```

Для пользователя с именем `simon` при подключении к любой базе данных:

```
ALTER ROLE saas
SET configuration_parameter = value2;
```

Для определенного пользователя, подключающегося к определенной базе данных:


```
ALTER ROLE simon  
IN DATABASE saas  
SET configuration_parameter = value3;
```

Пользователь не узнает, что такие настройки указаны только для него. Все эти параметры имеют значения по умолчанию, которые при необходимости можно перекрыть.

Как это работает

Можно назначать параметры:

- для базы данных;
- для пользователя (точнее, в PostgreSQL это «роль»);
- для комбинации «база данных / пользователь».

Для каждого параметра значение по умолчанию «перекрывается» значением, указанным ниже.

В предыдущих примерах:

- если пользователь `hannu` подключится к базе данных `saas`, то будет применено значение `value1`;
- если пользователь `simon` подключится к любой базе данных, кроме `saas`, то будет применено значение `value2`;
- если пользователь `simon` подключится к базе данных `saas`, то будет применено значение `value3`.

PostgreSQL воспринимает это так, как если бы пользователь сразу после установки соединения вручную вызвал эквивалент выражений `SET`.

Основной список параметров для проверки конфигурации сервера

PostgreSQL «из коробки» сконфигурирован для разделяемых систем, хотя многим удобнее работать с выделенным сервером. Настройки по умолчанию были оптимизированы таким образом, чтобы PostgreSQL не сражался за ресурсы с другими программами, запущенными на сервере, и чтобы не занимал все доступные ресурсы сервера. Если вы как системный администратор твердо уверены, что никакого другого важного программного обеспечения на сервере с PostgreSQL запущено не будет, можете спокойно увеличивать значения параметров.

Подготовка

Прежде всего необходимо выяснить две вещи:

- какой объем физической оперативной памяти можно будет выделить для PostgreSQL;
- приложения какого типа будут использовать PostgreSQL.

Как это сделать

Если размер вашей базы данных больше 32 МБ, возможно, следует увеличить значение параметра `shared_buffers`. Этому параметру допускается присваивать довольно большое значение, однако следует помнить, что в Linux-системах неиспользуемая память будет высвобождаться. Так что лучше быть консервативным и увеличивать этот параметр в `postgresql.conf` постепенно, каждый раз убеждаясь, что внесенные изменения оказываются выгодными.

Если вы определили большее значение для параметра `shared_buffers`, то на любом сервере, не являющемся Windows-сервером, вам наверняка придется увеличить значение параметра ОС `SHMMAX` (а на некоторых платформах и других параметров).

В Linux/Mac OS / FreeBSD вам также придется отредактировать файл `etc/sysctl.conf` или использовать `sysctl -w` со следующими значениями:

- Linux: `kernel.shmmax=value`;
- Mac OS: `kern.sysv.shmmax=value`;
- FreeBSD: `kern.ipc.shmmax=value`.

За более подробной информацией обращайтесь на сайт <http://www.postgresql.org/docs/8.4/static/kernel-resources.html#SYSVIPC>.

Например, в Linux добавьте в `/etc/sysctl.conf` следующую строку:

```
kernel.shmmax=value
```

Не стоит беспокоиться о параметре `effective_cache_size`. Он не относится к сверхважным, так что можно не тратить время на подбор его значения.

Если вам приходится много записывать, то можно сделать `wal_buffers` гораздо больше значения по умолчанию.

Если требуется много записывать и/или подгружать большие объемы данных, увеличьте значение `checkpoint_segments`.

Если в базе данных много больших запросов, можно увеличить значение `work_mem`.

Убедитесь в том, что включен `autovacuum`, если только у вас нет веской причины держать его выключенным. Как правило, такой

причины нет. Подробнее об `autovacuum` вы узнаете в следующих главах.

Хотите, чтобы было еще проще, – пройдите по ссылке <http://pgfoundry.org/projects/pgtune/>.

И на этом можно остановиться. Не стоит переживать из-за точных значений каких-либо параметров. Большую их часть вы измените позже, улучшая все постепенно.

Усвойте самое основное, а когда вам станет все просто и понятно, купите книгу Грегга Смита (Greg Smith) о производительности PostgreSQL.

И не трогайте параметр `fsync`. Так будет безопаснее!

Добавление внешних модулей в PostgreSQL

Среди сильных сторон PostgreSQL можно назвать расширяемость. Она планировалась при разработке СУБД еще в 1980-х. В настоящее время для PostgreSQL есть множество дополнительных модулей, подключаемых к ядру сервера PostgreSQL.

Дополнительные модули бывают нескольких типов:

- функции;
- типы данных;
- операторы;
- индексы.

Заметим, что многие инструменты и клиентские интерфейсы не требуют для работы с PostgreSQL специальной установки. Здесь же обсудим модули, которые расширяют и изменяют поведение сервера за рамками стандартных для SQL формы записи и функций.

Подготовка

Сначала выберите необходимый модуль.

В PostgreSQL нет автоматической системы управления пакетами, поэтому модули могут находиться в нескольких местах. Например:

- contrib-модули являются официальными дополнениями к «ядру» PostgreSQL. Документация по ним находится по адресу <http://www.postgresql.org/docs/9.0/static/contrib.html>;
- <http://pgfoundry.org/> – сайт, специально созданный разработчиками открытого ПО, чтобы делиться инструментарием

и модулями PostgreSQL. Здесь используется то же ПО, что и на SourceForge.net;

- на сайте <http://www.postgis.org/> вы найдете большие внешние проекты, такие как PostGIS, которые являются сложными модулями PostgreSQL.

Как это сделать

В некоторых случаях, если вы используете специальный установщик, такой как *OneClick*, модули можно добавлять прямо при установке.

В других случаях возможна установка из пакета. Например, как устанавливать модуль для совместимости с Oracle, описывается на странице http://www.postgres.cz/index.php/Oracle_functionality.

Сначала нужно загрузить его со страницы <http://pgfoundry.org/frs/download.php/2420/orafce-3.0.1-1.pg82.rhel5.i386.rpm>. Затем – установить, используя примерно такие команды:

```
rpm -ivh orafce-3.0.1-1.pg90.rhel5.i386.rpm
sudo apt-get install postgresql-8.4-orafce
```

Зачастую полезные модули не собраны в пакет, и тогда необходимо устанавливать их вручную. Это несложно и послужит хорошим упражнением: вы сможете лучше понимать, что происходит.

У каждого модуля свои требования к установке. Но есть два основных аспекта:

- установка SQL-объектов для этого модуля;
- установка динамически подгружаемых библиотек для этого модуля.

Для большинства полезных модулей требуется и то, и другое, но есть пара случаев, когда у модуля есть только динамически подгружаемые библиотеки (например, AutoExplain).

Затем нужно построить библиотеки, следуя инструкциям по установке модуля.

Установите библиотеку туда, где сервер сможет ее найти:

```
shared_preload_libraries = '$libdir/modlib'
Create the database objects
psql -d dbname -f SHAREDIR/contrib/module.sql
```

Как это работает

PostgreSQL может динамически подгружать библиотеки тремя разными способами:

- если во время сессии воспользоваться командой `LOAD`;
- если при запуске сервера выставить в `postgresql.conf` параметр `shared_preload_libraries`;
- если при открытии сессии назначить для определенного пользователя параметр `shared_preload_libraries` в `ALTER ROLE`.

Функции и объекты PostgreSQL могут ссылаться на код в этих библиотеках, что позволяет плотно привязать дополнительные модули к работе сервера. Благодаря такой тесной связи этот метод допускается использовать даже в приложениях, требующих высокой производительности, и разница между встроенными и добавленными возможностями не ощущается.

Запуск сервера в энергосберегающем режиме

Вопрос потребления электроэнергии крайне важен. Многие беспокоятся об окружающей среде, и пользователи PostgreSQL не являются исключением.

Подготовка

Если сервер PostgreSQL не должен работать непрерывно и у него бывают периоды простоя, вам могут пригодиться эти советы. Речь идет, например, о ноутбуке или о каком-либо неактивном виртуальном сервере.

Как это сделать

PostgreSQL – это серверная СУБД, которая бездействует, когда нет активных клиентов. Минимизировать активность сервера можно с помощью следующих параметров в файле `postgresql.conf`:

```
autovacuum = off;  
wal_writer_delay = 10000;  
bgwriter_delay = 10000.
```

Эти настройки можно выставить в то время, когда на сервере спойно. Как только на сервере появится много пользователей, их нужно будет сбросить до предыдущих значений.

Как это работает

Вот список некоторых процессов, постоянно находящихся в активном состоянии в ожидании, что в любой момент могут появиться клиенты:

- процесс записи (background writer, или процесс фоновой записи);
- процесс записи в WAL;
- архиватор, который находится в активном состоянии, если включено архивирование WAL;
- процесс получения WAL, который активен, если используется потоковая репликация;
- процесс автовакууминга.

Процесс фоновой записи по умолчанию активируется каждые 200 мс. Максимальный интервал может быть 10 с, и это довольно мало. Процесс можно отключить, указав `bgwriter_lru_maxpages = 0`.

Процесс записи в WAL по умолчанию активизируется каждые 200 мс. Максимальный интервал равен 10 с. Отключить этот процесс нельзя, однако, если записывать будет нечего, его работа сведется к периодическому включению и проверке.

Архиватор активируется через каждые 15 с и проверяет, производилась ли запись в файлы WAL. Его работа затрагивает директорию файловой системы. Это время не задается параметром, который можно было бы изменить.

Процесс получения WAL активизируется каждые 100 мс, проверяет, пришли ли новые данные репликации, и если нет, спит дальше. Вышеуказанное время изменить нельзя.

Автовакууминг по умолчанию активизируется каждые 60 с. Можно изменить это время с помощью `autovacuum_naptime` или совсем отключить автовакууминг, указав `autovacuum = off`.

Итак, если вы используете потоковую репликацию, сервер будет просыпаться каждые 100 мс. Если нет – время 200 мс (по умолчанию) можно увеличить до 10 с.



ГЛАВА 4.

Контроль над сервером

Рассматриваемые темы:

- Запуск сервера БД вручную
- Быстрая и безопасная остановка сервера
- Экстренная остановка сервера
- Перезагрузка конфигурационных файлов
- Быстрый перезапуск сервера
- Запрет на создание новых соединений
- Создание ограничения по одной сессии на пользователя
- Принудительное отключение пользователей
- Проектирование с учетом multi-tenancy
- Использование множественных схем
- Выделение для пользователя собственной базы данных
- Запуск нескольких серверов на одной системе
- Настройка пула соединений

Введение

PostgreSQL состоит из нескольких серверных процессов, главный из которых называется `postmaster`. Запуск сервера фактически является запуском этих процессов, а остановка сервера — их прерыванием.

Каждый `postmaster` ожидает от клиентов запросов на соединение к одному заданному порту. Каждый порт слушает только один `postmaster`, поэтому номер порта можно использовать в качестве уникального идентификатора процесса `postmaster` и сервера, которым он управляет.

При запуске сервера вы ссылаетесь на *директорию данных*, в которой находятся данные БД. Кроме основной директории данных могут быть вспомогательные табличные пространства, поэтому директория данных является основным, но не единственным местом, где могут размещаться данные, которыми управляет сервер. У каждого запущенного сервера своя директория данных, и каждая директория

данных может управляться не более чем одним работающим сервером (экземпляром).

Для любого действия с сервером БД необходимо знать директорию данных этого сервера. Основными такими действиями являются запуск и остановка сервера. К ним можно также добавить перезапуск, хотя на самом деле это просто последовательно выполненные остановка и запуск. Кроме того, есть перезагрузка сервера, которая означает перезагрузку конфигурационных файлов сервера.

Следует также упомянуть несколько важных моментов. Для PostgreSQL по умолчанию выделен порт 5432. Это закреплено в *Internet Assigned Numbers Authority (IANA)* и действует практически везде. Так как у каждого сервера PostgreSQL должен быть свой порт, для дополнительных серверов принято использовать подряд все последующие номера: 5433, 5434 и т. д. Эти номера уже не являются столь узнаваемыми, что может усложнить вам жизнь, если придется иметь дело с повышенными требованиями к безопасности в больших проектах.

Сервер БД иногда называют кластером, хотя этот термин не совсем удачен, поскольку наводит на мысль о множественных узлах, а не о единичном сервере.

Запуск сервера БД вручную

Как правило, сервер PostgreSQL стартует автоматически при загрузке операционной системы. Однако для вас, возможно, предпочтительней будет включать и отключать сервер вручную. Кроме того, ручное включение может понадобиться по каким-либо другим причинам.

Подготовка

Для начала выясним разницу между сервисом и сервером. Словом «сервер» обозначают сервер базы данных и его процессы. Слово «сервис» относится к обертке операционной системы, с помощью которой вызывается сервер. Сервер в основном на всех платформах работает одинаково, а вот сервисы могут отличаться как в разных операционных системах, так и в разных версиях операционных систем.

Как это сделать

В каждой операционной системе сервер запускается специальной командой:

UBUNTU/DEBIAN	<code>pg_ctlcluster 9.0 main reload</code>
RED HAT/FEDORA	<code>pg_ctl -D /var/lib/pgsql/data start</code>
SOLARIS	<code>pg_ctl -D /var/lib/pgsql/data start</code>
MAC OS	<code>pg_ctl -D /var/lib/pgsql/data start</code>
FREEBSD	<code>pg_ctl -D /var/lib/pgsql/data start</code>

хотя на некоторых платформах сервер можно запустить и другим способом:

RED HAT/FEDORA	<code>service postgresql start</code>
WINDOWS	<code>net start postgres</code>

Как это работает

В Ubuntu/Debian есть удобная утилита `pg_ctlcluster`, облегчающая одновременную работу нескольких серверов, что особенно полезно, если серверы имеют разные версии.

Быстрая и безопасная остановка сервера

Поехали!

Как это сделать

Сервер БД можно остановить в быстром режиме:

```
pg_ctl -D datadir -m fast stop
```

Если нужно остановить сервер как можно быстрее, необходимо использовать `-m fast`. При обычной остановке сервер ждет, пока все пользователи закончат работу. На это может уйти слишком много времени, даже если новые соединения будут запрещены.

В других системах эта команда может иметь вид:

```
DEBIAN/UBUNTU    pg_ctlcluster 9.0 main stop --force
```

Как это работает

При такой быстрой остановке все транзакции пользователей и все соединения обрываются. Это не слишком вежливо по отношению к пользователям, зато безопасно для сервера и его данных, что важнее.

Подобно другим СУБД, PostgreSQL перед отключением сервера сохраняет контрольную точку, что в дальнейшем позволяет быстро и легко запустить сервер. Чем больше информации требуется сохранить в контрольной точке, тем дольше сервер будет отключаться.

Но, в отличие от других СУБД, таких как Oracle, DB2 или SQL-Server, откат транзакций в PostgreSQL происходит очень быстро. В других системах выключение в режиме, требующем отката транзакций, может занять довольно много времени. Такая разница возникла по внутренним причинам, и в любом случае ничего опасного здесь нет.

В Debian/Ubuntu есть очень хорошая опция `--force`, которая сначала пытается выполнить быстрое отключение, а если это не удается, выполняет мгновенное отключение и затем останавливает процесс `postmaster`.

Экстренная остановка сервера

Если ничего не работает, нужно быстро остановить сервер. Осталось найти экстренную кнопку.

Как это сделать

Экстренный перезапуск сервера вызывается такой командой:

```
pg_ctl -D datadir stop -m immediate
```

Необходимо использовать режим `immediate stop`.

Как это работает

При мгновенной остановке сервера обрываются все транзакции и все соединения пользователей. Такая остановка не будет ни вежливой, ни «чистой».

Режим `immediate stop` похож на «падение» сервера. При перезапуске после такой остановки необходимо заново строить некоторые кэшированные файлы, а для базы данных в целом – пройти процедуру восстановления после «падения».

Для тех, кто работал с Oracle, заметим, что режим мгновенной остановки сервера в PostgreSQL – это то же самое, что *shutdown abort* в Oracle, но не то же самое, что *shutdown immediate*.

Перезагрузка конфигурационных файлов

Для смены некоторых конфигурационных параметров необходима перезагрузка всего конфигурационного файла.

Как это сделать

```

UBUNTU/DEBIAN    pg_ctlcluster 9.0 main reload
RED HAT/FEDORA    service postgresql reload
pg_ctl -D /var/lib/pgsql/data reload
SOLARIS           pg_ctl -D /var/lib/pgsql/data reload
MAC OS            pg_ctl -D /var/lib/pgsql/data reload
FREEBSD           pg_ctl -D /var/lib/pgsql/data reload

```

Конфигурационные файлы можно перезагрузить, не разрывая соединения с PostgreSQL, из командной строки. Для этого у вас должны быть права суперпользователя:

```

postgres=# select pg_reload_conf();
pg_reload_conf
-----
t

```

Данную команду можно выполнить с помощью какого-либо инструмента администрирования, например pgAdmin3.

Если вы поступите подобным образом, может оказаться, что было добавлено новое правило аутентификации, которому не удовлетворяет ваша сессия. В этом случае соединение не будет разорвано немедленно, однако в случае разрыва, возможно, вы не сможете создать новое.

Как это работает

Для перезагрузки конфигурационных файлов процессу postmaster посылается сигнал SIGHUP, который затем передается дальше по всем соединениям. Поэтому многие для обозначения этого процесса используют слово *sigh-up-ing* (*sigh-up-ирование*).

В каталожной таблице `pg_settings` есть столбец `context`. Каждый параметр можно менять в определенном месте и в определенный момент времени. Некоторые настройки можно сбросить только при перезагрузке сервера, поэтому в столбце `context` для них будет указано значение `sighup`. Возможно, вам понадобится время от времени менять эти настройки (есть и другие):

```

postgres=# SELECT name, setting, unit
, (source = 'default') as is_default
FROM pg_settings
WHERE context = 'sighup'
AND (name like '%delay' or name like '%timeout')
AND setting != '0';
name                | setting | unit | is_default
-----+-----+-----+-----
authentication_timeout | 60      | s    | t
autovacuum_vacuum_cost_delay | 20     | ms   | t

```

<code>bgwriter_delay</code>	10	ms	f
<code>checkpoint_timeout</code>	32	s	f
<code>deadlock_timeout</code>	1000	ms	t
<code>max_standby_delay</code>	30		t
<code>wal_sender_delay</code>	200	ms	t
<code>wal_writer_delay</code>	200	ms	t
(8 rows)			

И еще

В то время как перезагрузка конфигурационных файлов производится по сигналу SIGHUP, вы можете перегрузить конфигурационный файл для отдельного конечного пользователя с помощью команды `kill`. Полученный результат может оказаться неожиданным, так что делайте это с осторожностью.

Для начала найдите `pid` данного пользователя с помощью `pg_stat_activity`.

Затем выполните в командной строке операционной системы

```
kill -SIGHUP pid
```

Или можно совместить оба этих действия:

```
kill -SIGHUP `
psql -t -c "select procpid from pg_stat_activity limit 1"
```

Однако подобное решение имеет смысл только при использовании `WHERE`.

Быстрый перезапуск сервера

Для изменения некоторых параметров требуется полная остановка с последующим перезапуском сервера. Бывает, что сделать это нужно по возможности быстро. Лучше всего, конечно, в спокойное для сервера время, с тщательным планированием и тестированием, однако иногда планы приходится нарушать.

Как это сделать

Основная команда для перезапуска сервера:

```
pg_ctl -D datadir restart -m fast
```

Для перезапуска сервера необходимо просто остановить сервер, а затем повторно запустить его. С виду это довольно просто. Во многих случаях так оно и есть, но если необходимо перезапустить сильно загруженный сервер, то понадобится выполнить несколько действий, которые ускорят перезапуск.

Во-первых, остановка должна быть быстрой. Если вызвать остановку с настройками по умолчанию или «умную» остановку, сервер будет ждать, пока все закончат работу. Если выполнить немедленную остановку, сервер «упадет», и нужно будет восстанавливать потерянные данные, что в итоге займет еще больше времени.

Кэш работающего сервера заполнен блоками данных, многие из которых являются «грязными». Подобно другим СУБД, PostgreSQL перед закрытием создает контрольную точку. Таким образом, последующий старт будет быстрым и «чистым». Чем больше нужно записать в контрольной точке, тем больше времени займет отключение.

Отключение можно произвести гораздо быстрее, если сначала сохранить контрольную точку: это значительно ускорит сохранение контрольной точки при отключении. Итак, слейте все «грязные» `shared_buffers` на диск с помощью команды

```
psql -c "CHECKPOINT"
```

Далее можно предположить, что при повторном запуске системы кэш базы данных окажется пустым и потребуется время на его заполнение. Нормальное время разогрева – от 30 до 60 минут; только после этого вы полностью восстановите скорость работы. Так что непродуманный перезапуск системы может существенно повлиять на вашу работу.

Утилита `pg_cacheutils`, созданная одним из авторов этой книги, перед отключением сервера сохраняет содержимое кэша. Это позволяет быстро восстановить кэш и включиться в работу сразу после перезапуска.

```
psql -c "select pg_cache_save('mycache')"
```

Затем можно выполнить немедленное отключение сервера и «чисто» запустить его.

```
pg_ctl -D datadir -m immediate restart
```

После этого заполните кэш:

```
psql -c "select pg_cache_warm('mycache')"
```

Никакой магии здесь нет, поэтому придется немного подождать, пока работа не развернется на полную скорость.

Как это работает

Утилита `pg_cache_save()` сохраняет блоки данных в таблицу под названием `mycache`. Это позволит вам при необходимости сохранять различные наборы данных кэша для различных целей. Если не указать имя, по умолчанию будет использовано `saved_cache`.

При старте `pg_cache_warm()` прочитает сохраненную из кэша информацию и произведет выборку для каждого блока. Таким образом, по возможности вся информация будет возвращена в кэш. Несуществующие блоки будут проигнорированы.

Запрет на создание новых соединений

Иногда в экстренных случаях приходится полностью блокировать сервер или запрещать определенным пользователям доступ к базе данных. Все такие ситуации предсказать трудно, поэтому рассмотрим только некоторую часть.

Как это сделать

Соединения можно запретить одним из следующих способов:

- приостановить/возобновить пул сессии (подробнее в рецепте об управлении пулами соединений);
- остановить сервер (как это сделать, вы уже знаете, но лучше выбрать другой способ);
- ограничить для определенной базы данных число соединений до 0:

```
ALTER DATABASE foo_db CONNECTION LIMIT 0;
```

Этим вы ограничите доступ к базе данных для обычных пользователей, однако соединения для суперпользователей будут разрешены;

- ограничить число соединений для определенного пользователя до 0 (об этом вы прочитаете ниже):

```
ALTER USER foo CONNECTION LIMIT 0;
```

Этим ограничиваются соединения для обычного пользователя, однако если у него права суперпользователя, ограничение не сработает, так что сами себя вы выгнать не сможете;

- запретить в файле аутентификации по узлу (HBA – host-based authentication) все входящие соединения, а затем перезапустить сервер:
 - создайте новый файл `pg_hba_lockdown.conf` и добавьте в него две следующие строки. Таким образом, вы полностью заблокируете доступ к серверу, в том числе для

суперпользователей. Несомненно, это очень серьезные и радикальные действия:

#	TYPE	DATABASE	USER	CIDR-ADDRESS	METHOD
	local	all	all		reject
	host	all	all	0.0.0.0/0	reject

Если у вас все еще есть доступ суперпользователя, попробуйте что-то вроде этого:

#	TYPE	DATABASE	USER	CIDR-ADDRESS	METHOD
	local	all	postgres		ident
	local	all	all		reject
	host	all	all	0.0.0.0/0	reject

Таким образом, соединения будут запрещены для всех пользователей кроме системного пользователя `postgres`, который подключается к базе данных `postgres` локально. Не перепутайте второй и третий столбец: во втором указывается база данных, а в третьем – пользователь. Хотя бы ради этого стоит оставлять строку с заголовком. Если используется более сложная конфигурация, метод `ident` следует заменить на другой метод аутентификации;

- скопируйте текущий файл `pg_hba.conf` в `pg_hba_access.conf`, чтобы при необходимости восстановить его;
- скопируйте `pg_hba_lockdown.conf` в `pg_hba.conf`;
- перезагрузите сервер, пользуясь вышеописанным способом.

Как это работает

В файле `pg_hba.conf` определяются правила аутентификации по узлу. Речь идет не о самих аутентификациях, а об используемых для них механизмах. Эти правила являются главными для PostgreSQL. Они записаны в файле, и процесс `postmaster` применяет их при каждой попытке установить соединение. Причем неизвестно, является ли пользователь суперпользователем, так как о доступе к БД речь не идет. Таким образом, вы можете заблокировать *всех* пользователей. Доступ можно восстановить в любой момент; для этого необходимо отредактировать файл, а затем перезагрузить его.

Создание ограничения по одной сессии на пользователя

Если есть необходимость экономии ресурсов, можно ограничить пользователей так, чтобы они могли подключиться к серверу только

один раз. Для этого используется тот же метод, что и для полного запрета соединений.

Как это сделать

Можно создать ограничение в одно соединение на одного пользователя с помощью команды:

```
postgres=# ALTER ROLE fred CONNECTION LIMIT 1;  
ALTER ROLE
```

При этом при попытке повторно установить соединение будет выдаваться сообщение об ошибке:

```
FATAL: too many connections for role "fred".
```

Отменить это ограничение можно, указав значение -1.

Предел может быть равен 0 или любому положительному целому числу, причем это число может отличаться от `max_connections`, хотя это уже останется на вашей совести.

Указав значение, равное 0, вы запретите все соединения для обычных пользователей, однако для суперпользователей это не сработает.

Как это работает

Ограничение количества соединений вступает в силу при открытии сессии. Если вы увеличите это значение, на уже подсоединенных пользователей это не произведет никакого эффекта. Понижение также ни на что не будет влиять, до того как соединение будет разорвано и пользователь попытается вновь его установить.

Поэтому, если вы уменьшили предельное значение, необходимо проверить, нет ли «лишних» соединений, иначе в случае «падения» сервера могут возникнуть неприятные сюрпризы.

```
postgres=> SELECT rolconnlimit  
FROM pg_roles  
WHERE rolname = 'fred';  
rolconnlimit  
-----  
1  
(1 row)  
postgres=> SELECT count(*)  
FROM pg_stat_activity  
WHERE username = 'fred';  
count  
-----  
2  
(1 row)
```


Если после уменьшения числа допустимых соединений вы обнаружите «лишние» соединения, вы можете либо вежливо попросить пользователей отключиться либо воспользоваться инструкциями из следующего раздела данной книги.

И не беспокойтесь, пользователи самостоятельно не могут изменять лимит числа допустимых соединений.

Принудительное отключение пользователей

Иногда необходимо по какой-либо причине отключить от сервера целую группу пользователей.

Как это сделать

Вы можете прервать сессию пользователя с помощью встроенной функции `pg_terminate_backend()`. Эта функция берет `pid` или `id` процесса сессии пользователя на сервере. Речь идет именно о серверной части связки «клиент-сервер», а не о клиентском приложении.

Идентификатор `pid` пользователя можно узнать из представления `pg_stat_activity`. В запросе это будет выглядеть так:

```
SELECT pg_terminate_backend(procpid)
FROM   pg_stat_activity
WHERE  ...
```

При использовании этого запроса следует учитывать:

- если ни одна сессия не будет удовлетворять условиям, указанным в `WHERE`, то в результате вы ничего не получите. Аналогично, если условиям будет удовлетворять несколько сессий, результат окажется бесполезным;
- если по неосторожности включить в запрос собственную сессию, вы сами себя отключите! Причем произойдет это посредди отключения других пользователей, поскольку функция `pg_terminate_backend()` будет вызываться по мере выявления подходящих сессий.

Таким образом, лучше использовать более безопасный запрос:

```
postgres=# SELECT count(pg_terminate_backend(procpid))
FROM pg_stat_activity
WHERE username NOT IN
      (SELECT username
       FROM pg_user
```

```
WHERE usesuper);  
count  
-----  
1
```

Предполагается, что суперпользователи выполняют задачи администрирования.

Также можно использовать в качестве фильтров:

- WHERE application_name = 'myappname'
- WHERE waiting
- WHERE current_query = '<IDLE> in transaction'
- WHERE current_query = '<IDLE>'

Как это работает

Функция `pg_terminal_backend()` посылает сигнал непосредственно процессу операционной системы, работающему с этой сессией. Возможно, названная сессия уже была закрыта, поскольку `pid` назначается операционной системой, и затем могла быть открыта новая сессия с тем же `pid`, которую вы теперь и отключите. Так что следует быть внимательным.

Кроме того, с тех пор как был получен список активных сессий, могли быть открыты новые. Единственный способ избежать этого описан в разделе «Запрет на создание новых соединений».

Проектирование с учетом multi-tenancy

Существует множество причин, по которым нужно разделить группы таблиц или приложений: безопасность, контролирование ресурсов, удобство. Как бы то ни было, требуется отделить группы таблиц (мы сознательно избегаем термина «база данных» во избежание путаницы). Этот процесс часто называют multi-tenancy, хотя до сих пор термин не стал общепринятым.

Как это сделать

Если вы хотите запустить несколько физических баз данных на одном сервере, у вас есть четыре возможности:

1. Работать с несколькими наборами таблиц в разных схемах в одной базе данных PostgreSQL.

2. Работать с несколькими базами данных в одном экземпляре PostgreSQL.
3. Запустить несколько экземпляров PostgreSQL на одной виртуальной/физической системе.
4. Запустить несколько экземпляров PostgreSQL на отдельных виртуальных машинах на одном физическом сервере.

Пункт 1 описан в рецепте «Использование множественных схем».

Пункт 2 описан в рецепте «Выделение для пользователя собственной базы данных».

Пункт 3 описан в рецепте «Запуск нескольких серверов на одной системе».

Какой способ является самым лучшим? Сколько людей, столько и мнений. Ответ зависит от дополнительных требований:

- если необходимо разделить физические ресурсы, лучше всего подходят 3-й и 4-й пункты. Для отдельных серверов БД можно выделить отдельные диски и отдельные сегменты оперативной памяти – таким образом запуск и «падение» одного сервера не будут влиять на остальные;
- если вашей целью является безопасность, достаточно пункта 2;
- если необходимо просто разделить таблицы для удобства администрирования, подойдет как 1-й, так и 2-й пункт.

Пункт 2 позволяет полностью разделить таблицы, что полезно с точки зрения безопасности. Кроме того, тот, кто имеет привилегии доступа к обеим группам таблиц, не сможет их соединить. Таким образом, если впоследствии понадобится кросс-анализ, лучше воспользоваться пунктом 1, хотя и тут можно возразить, что такой анализ нужно производить в отдельном хранилище данных, а не связывая две «боевые» системы.

С выполнением пункта 3 в большей части дистрибутивов PostgreSQL есть небольшая трудность. По умолчанию при установке используется единственная локация для баз данных, что затрудняет конфигурацию в этом случае. В Ubuntu/Debian данная проблема решается легко, что делает их оптимальными системами для запуска нескольких экземпляров PostgreSQL на одной виртуальной/физической системе.

Пункт 4 реализуется с помощью технологии виртуализации, рассмотрение которой не входит в задачу данной книги.

Некоторые администраторы поддерживают на PostgreSQL тысячи баз данных, хотя в основном всем хватает одной БД postgres (ну или еще нескольких). Встречаются также системы с огромным количеством схем.

Следует понимать, что практически любые инструменты графического администрирования окажутся бесполезными при работе с сотнями или тысячами элементов. Как правило, в таких инструментах используется древовидная схема отображения, которая просто не справится с такими объемами.

Использование множественных схем

Группы таблиц можно отделить в их собственные *именные пространства* (namespaces), которые в PostgreSQL называются *схемами*. Они похожи на директории, но все же есть и различия.

Подготовка

Обязательно прочитайте раздел «Проектирование с учетом multi-tenancy», чтобы убедиться, что вам нужен именно этот способ действий. Существуют и другие возможности, которые в вашем случае могут оказаться более полезными.

Как это сделать

Схемы легко создать с помощью команд

```
CREATE SCHEMA finance;  
CREATE SCHEMA sales;
```

Затем можно будет создавать объекты непосредственно внутри схемы, используя «полные» имена, например:

```
CREATE TABLE finance.month_end_snapshot (...)
```

По умолчанию объект будет создаваться в текущей схеме. Узнать о том, какая схема в данный момент является текущей, можно с помощью запроса

```
postgres=# select current_schema;  
current_schema  
-----  
public  
(1 row)
```

При работе с объектами БД используется настраиваемый для различных пользователей параметр `search_path`, который определяет, в какой схеме следует искать. Текущая схема – это первая схема в `search_path` (отдельного параметра для текущей схемы нет).

Если необходимо, чтобы пользователь видел только определенный набор таблиц, можно модифицировать его `search_path`. Так как этот параметр для каждого пользователя может иметь свое значение, он задается при попытке установить соединение. Например, так:

```
ALTER ROLE fiona SET search_path = 'finance';  
ALTER ROLE sally SET search_path = 'sales';
```

Заметим, что схема `public` не упомянута в `search_path`, и поиск по ней проводиться не будет. Все таблицы, созданные пользователем `fiona`, будут по умолчанию помещены в схему `finance`, в то время как все, что создаст `sally`, по умолчанию отправится в схему `sales`.

Пользователи `finance` и `sales` смогут увидеть, что есть другая схема, однако можно так настроить привилегии, что пользователи одной схемы не смогут ни создавать объекты, ни читать данные в другой схеме.

```
REVOKE ALL ON SCHEMA finance FROM public;  
GRANT ALL ON SCHEMA finance TO fiona;  
REVOKE ALL ON SCHEMA sales FROM public;  
GRANT ALL ON SCHEMA sales TO sally;
```

Другой метод – разрешить одному пользователю давать привилегии только для одной схемы, но не права работы во всех остальных схемах. Это можно сделать следующим образом:

```
REVOKE ALL ON SCHEMA finance FROM public;  
GRANT USAGE ON SCHEMA finance TO public;  
GRANT CREATE ON SCHEMA finance TO fiona;  
REVOKE ALL ON SCHEMA sales FROM public;  
GRANT USAGE ON SCHEMA sales TO sally;  
GRANT CREATE ON SCHEMA sales TO sally;
```

Следует заметить, что необходимо давать привилегии для работы со схемой так же, как и особые права на объекты схемы. Поэтому для объектов нужно будет указывать, например:

```
GRANT SELECT ON month_end_snapshot TO public;
```

или задавать привилегии по умолчанию, которые будут использованы при создании объектов:

```
ALTER DEFAULT PRIVILEGES FOR USER fiona IN SCHEMA finance  
GRANT SELECT ON TABLES TO PUBLIC;
```

Как это работает

Выше уже говорилось о том, что схемы в чем-то похожи на директорию.

Понятие `search_path` в PostgreSQL напоминает концепцию временной окружения `PATH`. Текущая схема напоминает текущую

рабочую директорию. Здесь нет команды `cd` для смены директории, но зато есть `search_path`. Есть и еще несколько отличий: например, схемы PostgreSQL не ранжируются по иерархии, подобно директориям файловой системы.

Многие создают пользователя и схему с одним именем, чтобы работать, как в других СУБД, например в Oracle.

Следует заметить, что обе схемы `finance` и `sales` существуют в рамках одной и той же базы данных PostgreSQL и запущены на одном и том же сервере БД. Они используют один и тот же буфер, и их тесно связывает множество глобальных настроек.

Выделение для пользователя собственной базы данных

Отделение данных и пользователей является ключевой задачей администрирования. Всегда может возникнуть необходимость выделения для пользователя частной, защищенной или просто свободной от риска области («песочницы»). Вот как это делается.

Подготовка

Обязательно прочитайте рецепт «Проектирование с учетом *multi-tenancy*», чтобы убедиться, что вам нужен именно этот способ действий. Существуют другие возможности, которые в вашем случае могут оказаться более полезными.

Как это сделать

Создать базу данных для определенного пользователя очень просто. Если у вас есть права суперпользователя, наберите в командной строке:

```
postgres=# create user fred;  
CREATE ROLE  
postgres=# create database fred owner = fred;  
CREATE DATABASE
```

Как владелец базы данных пользователь имеет привилегию `login`, поэтому может по умолчанию подключаться к любой базе данных. Хотя есть команда `ALTER DEFAULT PRIVILEGES`, которая не применяется к базам данных, табличным пространствам или языкам, а только к существующим ролям (то есть к пользователям).

Так что необходимо отобрать привилегию подключаться к вашей базе данных у всех, кроме определенного пользователя. Команды `REVOKE ... FROM PUBLIC EXCEPT` нет, поэтому придется сначала отнять все права, а затем вернуть те, которые нужно:

```
postgres=# BEGIN;
BEGIN
postgres=# REVOKE connect ON DATABASE fred FROM public;
REVOKE
postgres=# GRANT connect ON DATABASE fred TO fred;
GRANT
postgres=# COMMIT;
COMMIT
postgres=# create user bob;
CREATE ROLE
```

Теперь попробуйте подключиться к базе данных `fred` как пользователь `bob`:

```
os $ psql -U bob fred
psql: FATAL: permission denied for database "fred"
DETAIL: User does not have CONNECT privilege.
```

что и требовалось получить.

Как это работает

PostgreSQL в большинстве случаев разрешает транзакционную DDL, поэтому в вышеприведенном примере либо сработают и `REVOKE`, и `GRANT`, либо ничего не произойдет. Таким образом, пользователь `fred` в любом случае не потеряет возможность подключаться к своей базе данных. Заметим, что `CREATE DATABASE` не может быть частью транзакции, хотя ничего серьезного из-за этого не случится.

И еще

Заметим, что суперпользователи смогут подключаться к этой новой базе данных и запретить им это невозможно.

Другие пользователи не смогут видеть ни таблицы, создаваемые в новой базе данных, ни имена ее объектов. Им будет видно только, что существует такая база данных, а также имя ее владельца.

Смотрите также

Подробнее об этом рассказывается в шестой главе.

Запуск нескольких серверов на одной системе

При необходимости вы можете запускать на одной физической системе несколько серверов PostgreSQL.

Подготовка

Обязательно прочитайте рецепт «Проектирование с учетом multi-tenancy», чтобы убедиться, что вам нужен именно этот способ действий. Существуют другие возможности, которые в вашем случае могут оказаться более полезными.

Как это сделать

Ядро PostgreSQL позволяет запускать сразу несколько серверов на одной системе. Однако есть несколько мелочей, которые следует иметь в виду.

Некоторые версии установщика создают директорию данных PostgreSQL с именем `data`. Поэтому могут возникнуть трудности с созданием двух и более директорий данных без изменения структуры директорий и имен.

Вариант для Debian/Ubuntu специально был разработан с учетом возможности одновременного запуска нескольких серверов и даже серверов различных версий.

Для создания дополнительной директории данных выполните

```
sudo pg_createcluster 9.0 database2
```

которую затем можно будет запускать таким образом:

```
sudo pg_createcluster 9.0 database2
```

При этом будет создан кластер БД версии 9.0 с именем `database2`, файлы которого будут храниться в директории

```
/var/lib/postgresql/9.0/database2.
```

В Red Hat нужно будет запустить `initdb` и внимательно выбрать директорию, например:

```
initdb -D /var/lib/pgsql/database2
```

затем поменять значение порта в `postgresql.conf` и запустить сервер:

```
pg_ctl -D /var/lib/pgsql/database2 start
```

При этом будет создан дополнительный кластер БД с версией сервера по умолчанию, в котором файлы хранятся в директории


```
/var/lib/pgsql/database2.
```

Кроме того, можно настроить сервер с помощью `chkconfig`.

Как это работает

Серверы PostgreSQL контролируются с помощью `pg_ctl`. Все остальное – это в некотором роде обертка. Единственные препятствия для запуска нескольких серверов в рамках одной системы – размещение файлов и обговоренные названия. Все остальное просто. Таким образом, Debian/Ubuntu – это практически единственные системы, полностью приспособленные для запуска нескольких серверов.

Настройка пула соединений

Пул соединений – это набор сессий, которые уже подключены. Его можно использовать для уменьшения нагрузки или для переподключения.

Пул соединений можно задавать множеством способов, в зависимости от того, какое программное обеспечение вы используете. Вероятно, лучше всего применять программы пулов соединений «со стороны сервера», поскольку это работает для всех типов соединений.

Здесь следует подробнее остановиться на `pgbouncer`, который разрабатывался как очень легковесный пул соединений. Как понятно из названия, пул можно приостановить/возобновить для того, чтобы перезапустить или «откатить» сервер.

Подготовка

Во-первых, следует выбрать, где хранить файлы с параметрами, файлы журнала и файлы с `pid`.

Пул `pgbouncer` может одновременно работать с файлами нескольких серверов, однако это не слишком мудрое решение. Если ассоциировать файлы `pgbouncer` с одним сервером, управлять им будет гораздо проще.

Как это сделать

Сконфигурируйте `pgbouncer` по шагам.

1. Создайте файл `pgbouncer.ini`:

```
;
; pgbouncer configuration example
;
[databases]
```

```
postgres = port=5432 dbname=postgres
[pgbouncer]
listen_port = 6543
listen_addr = 127.0.0.1
admin_users = postgres
;stats_users = monitoring user_id
auth_type = trust
; put these files somewhere sensible
auth_file = users.txt
logfile = pgbouncer.log
pidfile = pgbouncer.pid
; required for 9.0
ignore_startup_parameters = application_name
server_reset_query = DISCARD ALL;
; default values
pool_mode = session
default_pool_size = 20
log_pooler_errors = 0
```

2. Создайте файл `users.txt`. Он должен содержать как минимум пользователей, упомянутых в `admin_users` и `stats_users`. Например:

```
"postgres" ""
```

Пул `pgbouncer` также поддерживает аутентификацию по `md5`. Для эффективного использования необходимо скопировать зашифрованные пароли с сервера БД.

Можно при создании `users.txt` просто скопировать информацию с сервера. Для этого воспользуйтесь скриптом для `psql`:

```
postgres=> \o users.txt
postgres=> \t
postgres=> SELECT <*>||rolname||> <>||rolpassword||>>
postgres-> FROM pg_authid;
postgres=> \q
```

3. Запустите `pgbouncer`:
4. Проверьте соединение – оно должно отвечать `reload`:

```
psql -p 6543 -U postgres pgbouncer -c "reload"
```

Кроме того, убедитесь, что значение параметра `max_client_conn` в `pgbouncer` не превышает значение параметра `max_connections` в `PostgreSQL`.

Как это работает

Пул соединений `pgbouncer` – это отличная программа. Набор ее возможностей был выверен очень тщательно, чтобы получившийся

продукт был простым, надежным и очень быстрым. `pgbouncer` не является многопоточным и работает в одном процессе и, следовательно, на одном процессоре. Он очень эффективен, но передача больших объемов данных может занять много времени и ухудшить согласованность, поэтому передачу таких данных лучше делать по прямому соединению.

Пул `pgbouncer` не поддерживает SSL-соединения. Если бы поддерживал, то вся работа по шифрованию/дешифровке проводилась бы в одном-единственном процессе, что не лучшим образом повлияло бы на производительность. Для защищенных соединений следует использовать `stunnel`.

Программа `pgbouncer` предоставляет `connection pooling`. Если указать

```
pool_mode = transaction
```

`pgbouncer` также предоставит концентрированное соединение. Это позволит обрабатывать сотни тысяч входящих соединений при небольшом количестве соединений с сервером.

По мере появления новых соединений/транзакций/выражений пул увеличивает размеры до указанных пользователем максимумов. До того как пул освободит их, эти соединения «подвисают» в `server_idle_timeout`.

Также `pgbouncer` выпускает сессии с интервалом `server_lifetime`. Это позволяет серверу высвобождать свою часть связки «клиент-сервер», чтобы поддержать ротацию и избежать появления слишком длительных соединений сессий.

И еще

Для выполнения команд можно подключиться к самому `pgbouncer`. Это можно сделать интерактивно, как если бы вы вошли в `psql`, либо с помощью отдельных команд или скриптов.

Для отключения сервера можно просто набрать `SHUTDOWN` или ввести отдельную команду:

```
psql -p 6543 pgbouncer -c "SHUTDOWN"
```

Можно также воспользоваться командой `RELOAD`, чтобы заставить сервер переподгрузить (то есть перечитать) файлы с параметрами.

Если вы применяете `pool_mode = transaction` или `pool_mode = statement`, можно воспользоваться командой `PAUSE`. Это позволяет закончить текущую транзакцию, а затем придержать всю работу сессии.

Таким образом, вы упростите выполнение DDL или перезапуск сервера.

Пул `pgbouncer` также позволяет использовать режим `SUSPEND`. При этом он подождет, пока будут «слиты» все буферы сервера.

После `PAUSE` или `SUSPEND`, когда проделана необходимая работа, можно выполнить `RESUME`.

И в дополнение к командам контроля `pgbouncer` посмотрите несколько команд `SHOW`.

Таблица 4.1.

Команда <code>SHOW</code>	Результат
<code>SHOW STATS</code>	Статистика трафика. Общая и средняя по запросам, длительности запроса, получено/отправлено байт
<code>SHOW SERVERS</code>	В каждой строке одно соединение с сервером БД
<code>SHOW CLIENTS</code>	В каждой строке одно соединение от клиента
<code>SHOW POOLS</code>	Один пользователь из пула пользователей
<code>SHOW LISTS</code>	Обзор использования ресурсов
<code>SHOW USERS</code>	Список пользователей в <code>user.txt</code>
<code>SHOW DATABASES</code>	Список баз данных в <code>pgbouncer.ini</code>
<code>SHOW CONFIG</code>	Список конфигурационных параметров
<code>SHOW FDS</code>	Описание файлов
<code>SHOW SOCKETS</code>	Сокеты файлов
<code>SHOW VERSION</code>	Версия <code>pgbouncer</code>



ГЛАВА 5.

Таблицы и данные

Рассматриваемые темы:

- Выбор имени для объекта БД
- Управление объектами с именами, заключенными в кавычки
- Принудительная установка единого имени, единого определения столбцов
- Обнаружение и удаление дубликатов
- Предотвращение появления строк-дубликатов
- Поиск уникального ключа для выборки данных
- Генерирование тестовых данных
- Случайные данные по примеру
- Подгрузка данных из электронных таблиц
- Подгрузка данных из файлов

Введение

В этой главе собраны рецепты для работы с таблицами и содержащимися в них данными. Во многих рецептах вы встретите общие советы, которые проиллюстрированы примерами, характерными для PostgreSQL.

Некоторые администраторы занимаются только внешней частью сервера БД, а что происходит внутри баз данных, их не особенно волнует. Вот вам совет: позаботьтесь о своих данных, и ваша БД позаботится о вас! Содержите данные в чистоте, и запросы будут выполняться быстрее, ошибок в приложениях будет меньше, и вы сможете завязать хорошие деловые контакты. Не очень приятно, когда среди ночи вам звонят и сообщают, что что-то сломалось и нужно срочно исправлять ошибки.

Выбор имени для объекта БД

Если вы хотите, чтобы другие могли разобраться в вашей базе данных, выбирайте для объектов понятные имена.

Как это сделать

Что же делает имя понятным? Следуйте приведенным ниже рекомендациям:

- имя должно соответствовать существующим стандартам, общим и внутренним. Изобретение своих стандартов здесь не поможет, а вот строгое следование существующим – обязательно;
- имя должно явно описывать роль или содержимое таблицы;
- для основных таблиц используйте короткие выразительные имена;
- справочные таблицы называйте с учетом имени таблиц, на которые они ссылаются, например `customer_account`;
- убедитесь в том, что имя отличается от других похожих имен;
- разумно используйте сокращения;
- используйте подчеркивания. Регистр букв не сохраняется, поэтому имена вида `customerAccount`, как в Java, здесь будут просто нечитаемыми. Посмотрите рецепт работы с объектами, имена которых заключены в кавычки;
- вдумчиво используйте множественное число. Или не используйте;
- используйте суффиксы для идентификации типа содержимого или области, к которой принадлежит объект. PostgreSQL всегда использует суффиксы в именах автоматических сгенерированных объектов;
- будьте дальновидны при выборе имен: имя не должно опираться на текущее положение объекта или роль. Не стоит давать таблице имя `London`, только потому что она находится на сервере, который стоит в Лондоне – впоследствии она запросто может оказаться в Лос-Анджелесе;
- думайте о будущем. Не давайте имена, подразумевающие, что объект является единственным в своем роде – например, не называйте таблицу `TEST` или `BACKUP_DATA`;
- не используйте акронимы вместо длинных имен таблиц. По сравнению с `MAD` гораздо лучше выглядит `money_allocation_decision`, тем более что регистр не сохраняется и через некоторое время нелегко будет догадаться, что означает это слово (которое уже не будет восприниматься как аббревиатура);
- имя таблицы зачастую используется в качестве корня для имен других объектов, поэтому не стоит добавлять суффикс `table` и т. п.

И еще

Индексы в PostgreSQL имеют стандартные имена:

```
(tablename)_(columnname(s))_(suffix)
```

где в качестве *suffix* может выступать:

- *pkey* для ограничения по первичному ключу;
- *key* – уникальное ограничение;
- *excl* – исключительное ограничение;
- *idx* – для любых других видов индексов.

Стандартным суффиксом для последовательностей является *seq*.

У таблиц могут быть триггеры, заряженные на каждое событие. Триггеры выполняются в алфавитном порядке, поэтому следует называть их с учетом оказываемых ими действий и необходимого порядка их выполнения. Допустимо, конечно, использовать в именах триггеров слова *INSERT*, *UPDATE* или *DELETE*, но может возникнуть путаница, если у вас есть триггеры, которые, к примеру, работают и с *UPDATE*, и с *DELETE*.

Общепринятым считается такое обозначение триггеров:

```
(tablename)_(actionname)_(after|before)___trig
```

Если вам встретятся странные и/или неправильные имена объектов, разумно будет воспользоваться командой *RENAME*. Например:

```
ALTER INDEX badly_named_index RENAME TO tablename_status_idx;
```

Управление объектами с именами, заключенными в кавычки

В именах объектов PostgreSQL могут содержаться пробелы и буквы в разных регистрах, но для этого необходимо заключить имена таблиц в двойные кавычки. Здесь могут возникнуть некоторые трудности, и в этом рецепте рассказывается, как их преодолеть.

С проблемой чувствительности к регистру часто сталкиваются те, кто ранее работал с такими СУБД, как MySQL, или те, перед кем стоит задача перехода с MySQL.

Подготовка

Для начала создадим таблицу с именем, заключенным в кавычки, в котором присутствуют заглавные и строчные буквы.

```
CREATE TABLE "MyCust"  
AS  
SELECT * FROM cust;
```

Как это сделать

При попытке получить доступ к такой таблице обычным образом будет выдано сообщение об ошибке:

```
postgres=# SELECT count(*) FROM mycust;  
ERROR: relation "mycust" does not exist  
LINE 1: SELECT * FROM mycust;
```

То же самое произойдет, если написать с учетом регистра:

```
postgres=# SELECT count(*) FROM MyCust;  
ERROR: relation "mycust" does not exist  
LINE 1: SELECT * FROM mycust;
```

Если нужно получить доступ к таблице, имя которой при создании было заключено в кавычки, необходимо в запросе также указывать имя в кавычках:

```
postgres=# SELECT count(*) FROM "MyCust";  
count  
-----  
5  
(1 row)
```

Таким образом, если вы используете кавычки в названиях таблиц, не забывайте их ставить при работе с этими таблицами. С другой стороны, если в SQL используются имена, заключенные в кавычки, вероятно, вам нужно будет создавать таблицы с использованием кавычек.

Как это работает

В SQL-выражениях PostgreSQL меняет в именах все заглавные буквы на строчные. Например,

```
SELECT * FROM mycust;
```

для PostgreSQL не будет отличаться от

```
SELECT * FROM MYCUST;
```

или

```
SELECT * FROM MyCust;
```

Однако такая запись

```
SELECT * FROM "MyCust";
```

уже будет иметь другое значение.

И еще

Если вы берете значения из таблицы, которая использовалась для создания имен объектов, вам может пригодиться функция `quote_ident()`. Она сама расставит кавычки в именах объектов там, где этого требует PostgreSQL:

```
postgres=# select quote_ident('MyCust');
quote_ident
-----
'MyCust'
(1 row)

postgres=# select quote_ident('mycust');
quote_ident
-----
mycust
(1 row)
```

Особенно эта функция может пригодиться, если вы создаете таблицу с переменным именем в функции PL/pgSQL:

```
EXECUTE 'CREATE TEMP TABLE ' || quote_ident(tablename) ||
'(coll INTEGER);'
```

Принудительная установка единого имени, единого определения столбцов

В БД, спроектированных разумно, определения простые и недвусмысленные. Пользователи могут легко понять значения в каждой таблице. Это хороший способ избежать проблем с качеством данных.

Подготовка

Если вы хотите попробовать выполнить запросы из этого рецепта в качестве теста, воспользуйтесь следующими примерами. Вы также можете проверить свою собственную базу данных:

```
CREATE SCHEMA S1;
CREATE SCHEMA S2;
CREATE TABLE S1.X
(col1 INTEGER
,col2 TEXT);
CREATE TABLE S2.X
(col1 INTEGER
,col3 NUMERIC);
```

Как это сделать

Столбцы

Столбцы, определенные различными способами в разных таблицах, можно найти, направив запрос к каталогу. Воспользуйтесь запросом к информационной схеме:

```
SELECT
table_schema
,table_name
,column_name
,data_type
||coalesce(' ' || text(character_maximum_length), '')
||coalesce(' ' || text(numeric_precision), '')
||coalesce(', ' || text(numeric_scale), '')
as data_type
FROM information_schema.columns
WHERE column_name IN
(SELECT
column_name
FROM
(SELECT
column_name
,data_type
,character_maximum_length
,numeric_precision
,numeric_scale
FROM information_schema.columns
WHERE table_schema = 'public'
GROUP BY
column_name
,data_type
,character_maximum_length
,numeric_precision
,numeric_scale
)derived
GROUP BY column_name
HAVING count(*) > 1
)
AND table_schema NOT IN ('information_schema', 'pg_
catalog')
ORDER BY column_name
;
```

И в результате получите:

table_schema	table_name	column_name	data_type
s2	x	coll	integer 32,0
s1	x	coll	smallint 16,0

(2 rows)

Таблицы

Сравнение двух таблиц гораздо сложнее, так как есть больше вариантов, по которым таблицы могут различаться или быть похожими друг на друга. Следующий запрос ищет все таблицы с одинаковыми именами (следовательно, в разных схемах), но с различными определениями:

```
SELECT
    table_schema
,table_name
,column_name
,data_type
FROM information_schema.columns
WHERE table_name IN
(SELECT
    table_name
FROM
(SELECT
    table_schema
,table_name
,string_agg(' '||column_name||' '||data_type)
FROM information_schema.columns
GROUP BY
    table_schema
,table_name
) def
GROUP BY
table_name
HAVING
    count(*) > 1
)
ORDER BY
table_name
,table_schema
,column_name;
```

В результате получаем:

table_schema	table_name	column_name	data_type
s1	x	col1	smallint
s1	x	col2	text
s2	x	col1	integer
s2	x	col3	numeric

(4 rows)

Как это работает

Определения таблицы находятся внутри PostgreSQL, и получить к ним доступ можно через каталожные представления информационных схем.

Определения могут отличаться по разным причинам. Исключим внутреннюю таблицы PostgreSQL, поскольку в разных каталогах у них могут быть похожие имена, – это стандартная информационная схема SQL, реализованная в PostgreSQL, и собственная внутренняя схема PostgreSQL `pg_catalog`.

Эти запросы уже и так достаточно сложны, но их можно еще больше усложнить, если добавить в сравнение такие параметры, как значения по умолчанию (`DEFAULT`) или ограничения. Взяв такой запрос за основу, вы можете расширять его в любую сторону.

И еще

Определения двух таблиц можно сравнивать с помощью функции:

```
CREATE OR REPLACE FUNCTION diff_table_definition
(t1_schemaname text
,t1_tablename text
,t2_schemaname text
,t2_tablename text)
RETURNS TABLE
(t1_column_name text
,t1_data_type text
,t2_column_name text
,t2_data_type text
)
LANGUAGE SQL
as
$$
SELECT
    t1.column_name
    ,t1.data_type
    ,t2.column_name
    ,t2.data_type
FROM
    (SELECT column_name, data_type
     FROM information_schema.columns
     WHERE table_schema = $1
       AND table_name = $2
    ) t1
FULL OUTER JOIN
    (SELECT column_name, data_type
     FROM information_schema.columns
     WHERE table_schema = $3
       AND table_name = $4
    ) t2
ON (t1.column_name = t2.column_name)
WHERE t1.column_name IS NULL OR t2.column_name IS NULL
;
$;
```

Обнаружение и удаление дубликатов

Принцип работы реляционных баз данных основан на том, что любые данные уникальны. Однако даже при всем старании вы не сможете избежать появления дублированных данных – они придут откуда-нибудь со стороны. В данном разделе будет показано, как это обнаружить и какие меры следует принять.

Подготовка

В качестве примера для начала рассмотрим таблицу `cust`, в столбце `customerid` которой обнаружился дубликат:

```
postgres=# SELECT * FROM cust;
customerid | firstname | lastname | age
-----+-----+-----+-----
          1 | Philip    | Marlowe   | 38
          2 | Richard  | Hannay    | 42
          3 | Holly    | Martins   | 25
          4 | Harry    | Palmer    | 36
          4 | Mark     | Hall      | 47
(5 rows)
```

Прежде чем удалять дубликат, подумайте о том, что, возможно, это не «плохие» данные – просто вы их так воспринимаете. В таком случае нужно переделать модель БД и в рассмотрении учитывать еще и дополнительные таблицы. К тому же столбцы-дубликаты могли образоваться в процессе подгрузки данных, если вы решили исключить какой-то столбец. В общем, прежде чем что-то удалять, стоит подумать дважды.

Как это сделать

Сначала необходимо обнаружить дубликаты:

```
SELECT *
FROM cust
WHERE customerid IN
  (SELECT customerid
   FROM cust
   GROUP BY customerid
   HAVING count(*) > 1);
```

Полученный результат можно обработать вручную, выполнив следующие шаги:

- при необходимости для лучшей картины слить две строки. При этом можно использовать значения из одной строки для той, которую вы хотите оставить:

```
UPDATE cust
SET age = 47
WHERE customerid = 4
AND lastname = 'Palmer';
```

- удалить лишнюю строку:

```
DELETE FROM cust
WHERE customerid = 4
AND lastname = 'Hall';
```

В некоторых случаях строки могут оказаться полностью идентичными, как в следующей таблице new_cust:

```
postgres=# SELECT * FROM new_cust;
 customerid
-----
          1
          2
          3
          4
          4
(5 rows)
```

На этот раз данные оказываются совсем неразличимыми, поэтому можно автоматически удалить одну из строк-дубликатов. SQL работает с выборками данных, поэтому взять только одну строку из нескольких не так просто, как хотелось бы. Для удаления всех строк-дубликатов, кроме одной, воспользуйтесь таким блоком запросов:

```
BEGIN;
LOCK TABLE new_cust IN ROW EXCLUSIVE MODE;
DELETE FROM new_cust
WHERE ctid NOT IN
  (SELECT min(ctid)
   FROM new_cust
   WHERE customer_id IN (4) --specify exact duplicate ids
   GROUP BY customerid);
COMMIT;
```

а затем выполните

```
VACUUM new_cust;
```

чтобы очистить таблицу после удалений.

Как это работает

Первый запрос группирует все строки по уникальным значениям в столбцах и считает их. Если в какой-то момент строк оказалось

больше одной, это может означать наличие дубликата. Если искать по двум столбцам и более, то запрос должен выглядеть так:

```
SELECT *
FROM mytable
WHERE (col1, col2, ... , colN) IN
      (SELECT col1, col2, ... , colN
       FROM mytable
       GROUP BY col1, col2, ... , colN
       HAVING count(*) > 1);
```

где (col1, col2, ... , colN) – это список ключевых столбцов.

Заметим, что при этом придется отсортировать таблицу по всем ключевым столбцам, для чего понадобится объем, примерно равный объему таблицы. Безусловно, такой запрос следует с осторожностью использовать в больших таблицах. Если размер таблицы 128 МБ или более, для его выполнения стоит увеличить параметр `work_mem`.

Представленный выше запрос `DELETE` будет работать только в PostgreSQL, поскольку в нем используется внутренний идентификатор строк `ctid`. Если вы планируете выполнить этот запрос для нескольких столбцов, необходимо будет модифицировать его следующим образом:

```
DELETE FROM mytable
WHERE ctid NOT IN
      (SELECT min(ctid)
       FROM has_duplicates
       -- need WHERE clause to filter only exact duplicates
       GROUP BY col1, col2, ... , colN);
```

Строки с похожими значениями группируются и сортируются по `ctid`. Строка с наименьшим значением `ctid` находится ближе всего к началу таблицы; она и будет оставлена. После удаления и выполнения `VACUUM` вы обнаружите, что размер таблицы уменьшился, поскольку все дубликаты в конце таблицы были удалены.

Команды `BEGIN` и `COMMIT` собирают команды `LOCK` и `DELETE` в одну транзакцию. Командой `LOCK` осуществляется блокировка таблицы на довольно высоком уровне, что запрещает проводить `UPDATE` и `DELETE`, пока не будут удалены все дубликаты.

И еще

Не всегда возможно заблокировать таблицу на большой промежуток времени, пока все дубликаты будут удалены. Особенно это критично для больших таблиц. В таком случае необходимо действовать немного по-другому:

1. Вы находите строки, которые являются кандидатами на удаление, и копируете их в отдельную таблицу.

2. Строите индекс основной таблицы для ускорения доступа к строкам.
3. Пишете программу, которая в цикле просматривает строки из отдельной таблицы и выполняет маленькие транзакции.
4. Открываете новую транзакцию.
5. Выбираете в сторонней таблице подходящую выборку строк.
6. Быстро находите их в основной таблице с помощью индекса.
7. Удаляете соответствующие строки.
8. Закрываете транзакцию и повторяете цикл.

Упомянутая программа не может быть функцией для базы данных, так как в функции не допускается несколько транзакций. Несколько транзакций необходимы для того, чтобы заблокировать каждую строку на минимально возможное время.

Предотвращение появления строк-дубликатов

Отсутствие дубликатов – это одно из самых важных требований к качеству данных. В этой области PostgreSQL, в отличие от многих других реляционных СУБД, предлагает несколько решений.

Подготовка

Выберите столбцы, которые необходимо сделать уникальными. Но сначала решите, все ли столбцы должны быть единственными в своем роде или только часть из них.

Начнем с такого примера

```
postgres=# SELECT * FROM newcust;  
customerid  
-----  
          1  
          2  
          3  
          4  
(4 rows)
```

Как это сделать

Для предотвращения дублирования нужен уникальный индекс, который сервер БД сможет использовать, чтобы сделать уникальными необходимые столбцы.

Такой индекс для обычных типов данных можно выбрать одним из трех способов:

1. Создать для выборки столбцов первичный ключ, который в таблице может быть только один. Теперь значения строк не могут быть определены как NULL, поскольку вы принудительно задаете значения столбцов NOT NULL (если это до сих пор не так).

```
ALTER TABLE newcust ADD PRIMARY KEY(customerid);
```

При этом создается новый индекс с именем `newcust_pkey`.

2. Создать ограничение UNIQUE для этой выборки столбцов. Такой столбец может быть использован вместо или в качестве первичного ключа, и их число в таблице неограниченно. Значения NULL допустимы.

```
ALTER TABLE newcust ADD UNIQUE(customerid);
```

При этом создается новый индекс с именем `newcust_customerid_key`.

3. Создать для выборки столбцов UNIQUE INDEX:

```
CREATE UNIQUE INDEX ON newcust (customerid);
```

При этом будет создан новый индекс с именем `newcust_customerid_idx`.

Все эти способы исключают дубликаты. Во всех трех случаях создается индекс, хотя только в первых двух появляется формальное ограничение. И можно действовать любым способом, если у вас есть первичный ключ или уникальное ограничение, использующее несколько столбцов.

Последний метод важен тем, что к индексу можно применить оператор `WHERE`. Это может быть полезным в ситуациях, когда значения данных в столбцах уникальны только при определенных обстоятельствах. Поэтому полученный индекс еще называют *частичным*.

Если бы у вас была такая таблица:

```
postgres=# select * from partial_unique;
customerid | status | close_date
-----+-----+-----
1 | OPEN | 
2 | OPEN | 
3 | OPEN | 
3 | CLOSED | 2010-03-22
(4 rows)
```

можно было бы применить частичный индекс, чтобы уникальность в столбце `customerid` требовалась только для `status = 'OPEN'`. Например:

```
CREATE UNIQUE INDEX ON partial_unique (customerid)
WHERE status = 'OPEN';
```

Если необходимо наложить ограничение уникальности на данные более сложных типов, то запись придется усложнить. Вам помогут нижеследующие примеры.

Начнем с самого простого: создадим таблицу из ячеек и поместим в них какие-нибудь данные. Если это ваша первая встреча с записью типов данных в PostgreSQL, устраивайтесь поудобнее и смотрите внимательно.

```
postgres=# CREATE TABLE boxes (name text, position box);
CREATE TABLE
postgres=# INSERT INTO boxes VALUES
              ('First', box '((0,0), (1,1))');
INSERT 0 1
postgres=# INSERT INTO boxes VALUES
              ('Second', box '((2,0), (2,1))');
INSERT 0 1
postgres=# SELECT * FROM boxes;
name      | position
-----+-----
First     | ((0,0),(1,1))
Second    | ((2,0),(2,1))
(2 rows)
```

Здесь две ячейки, не касающиеся друг друга и не пересекающиеся, заданные координатами (x, y).

Чтобы добиться уникальности, вы можете создать ограничение, при котором будут отклонены все попытки добавить позицию, каким-либо образом пересекающуюся с уже существующими. В качестве оператора пересечения определите `&&`, и таким образом вы добавите ограничение:

```
postgres=# ALTER TABLE boxes ADD EXCLUDE USING gist
              (position WITH &&);
NOTICE: ALTER TABLE / ADD EXCLUDE will create implicit
index "boxes_
position_exclusion" for table "boxes"
ALTER TABLE
```

При этом создается новый индекс с именем `boxes_position_excl`.

Можно было бы использовать ту же форму записи, что и для обычных типов данных. Итак, четвертый способ решения этой задачи:

```
ALTER TABLE newcust ADD EXCLUDE (customerid WITH =);
```

При этом создается новый индекс `newcust_customerid_excl`.

Как это работает

Уникальность всегда задается индексом. Каждый индекс определяется оператором типа данных, который при добавлении новой строки

или обновлении значений в столбцах проверяет, не нарушится ли уникальность.

Таким образом, для соблюдения уникальности необходимы индекс и оператор поиска, определенный для типов данных столбцов. Если вы задаете обычные ограничения `UNIQUE`, в качестве оператора будет использован оператор равенства (`=`). В случае с `EXCLUDE` запись для решения той же задачи будет сложнее.

И еще

Ограничения уникальности можно отложить, то есть присвоить этому параметру значение `DEFERRABLE`, однако есть некоторые нюансы, которые делают эту возможность в PostgreSQL 9.0 не такой уж полезной:

- необходимо указать, что ограничение будет `DEFERRABLE` при создании таблицы с помощью `CREATE TABLE`. В `CREATE TABLE AS SELECT` или с помощью `ALTER TABLE` этого сделать нельзя;
- откладываемые ограничения уникальности нельзя использовать вместе с внешними ключами. При попытке добавить внешний ключ, который ссылается на откладываемое ограничение, вы получите сообщение об ошибке.

Команда разработчиков PostgreSQL работает над этой проблемой, и в последующих релизах она, скорее всего, будет решена.

Дублированные индексы

Заметим, что в PostgreSQL допустимы индексы с совпадающими определениями. В каких-то случаях это может быть полезным, но может и раздражать, если вы случайно создали несколько индексов вместо одного. Вы также можете наложить на индексы ограничения одним из вышеперечисленных способов, однако следует иметь в виду, что все ограничения в таком случае будут значительно усилены.

Уникальность без индексов

Уникальности выборки столбцов можно добиться и без создания индекса. Это пригодится, если уникальность нужна, а времени на создание индекса нет.

Необходимо выполнить следующее:

- использовать тип данных `serial`;
- вручную указать в качестве значения по умолчанию `nextval()` последовательности.

И в том и в другом случае вы получите уникальное значение, которое можно использовать в качестве ключа для каждой строки. При этом уникальность не является вынужденной, и не накладываются никакие ограничения. То есть сохраняется возможность, что кто-то вернет последовательности более раннее значение, которое приведет к появлению дубликатов.

Возможно, вам подойдут почти уникальные данные, такие как значения функции `timeofday()`, показывающей возрастающее время в микросекундах.

Пример из жизни: выделение диапазонов IP-адресов

Нужно определить диапазоны IP-адресов; при этом один IP-адрес должен соответствовать только одному пользователю или задаче. Все просто, если отслеживать каждый IP-адрес, но задача усложняется, если вы работаете с целыми диапазонами.

Сначала необходимо спланировать, какой будет база данных:

```
CREATE TABLE iprange
(iprange_start inet
,iprange_stop inet
,owner text);
INSERT INTO iprange VALUES
('192.168.0.1','192.168.0.16', 'Simon');
INSERT INTO iprange VALUES
('192.168.0.17','192.168.0.24', 'Greg');
INSERT INTO iprange VALUES
('192.168.0.32','192.168.0.64', 'Hannu');
```

Но ведь нельзя создать ограничение уникальности, которое усилит ограничение. Можно написать триггер, который будет проверять существующие значения, но здесь и возникает путаница.

Скачайте со страницы <http://pgfoundry.org/projects/ip4r/> и установите модуль типа данных `ip4r` – и вы сможете решить эту задачу.

Затем создайте вот такую таблицу и заполните ее теми же данными, но немного в другой форме:

```
CREATE TABLE iprange2
(iprange ip4r
,owner text);
INSERT INTO iprange2 VALUES
('192.168.0.1-192.168.0.16', 'Simon');
INSERT INTO iprange2 VALUES
('192.168.0.17-192.168.0.24', 'Greg');
INSERT INTO iprange2 VALUES
('192.168.0.32-192.168.0.64', 'Hannu');
```

Теперь вы можете создать исключяющее ограничение для таблицы с помощью следующей команды:

```
ALTER TABLE iprange2
ADD EXCLUDE USING GIST (iprange WITH &&);
```

Пример из жизни: диапазон времени

Во многих БД могут быть таблицы с историей, у которых для данных есть `START_DATE` и `END_DATE` или нечто подобное. Это поддерживается другим внешним модулем PostgreSQL с помощью типа данных `period`. `period` — пара значений `TIMESTAMPZ`, которая позволяет обозначить начальную и конечную точку `timestampz` (дата/время). Модуль можно найти по ссылке <http://temporal.projects.postgresql.org/>.

Пример из жизни: диапазоны префиксов

Еще одна распространенная задача — присваивание номеров кредитных карт или номеров телефона. Например, в случае с кредитными картами необходимо проводить дополнительные проверки финансового состояния.

Для этого класса задач был разработан тип данных `prefix range`. Соответствующее дополнение к PostgreSQL вы найдете по ссылке <http://github.com/dimitri/prefix>.

Поиск уникального ключа для выборки данных

Иногда затруднительно найти уникальную выборку ключевых столбцов, описывающую данные.

Подготовка

Начнем с маленькой таблицы, где ответ очевиден:

```
postgres=# select * from ord;
orderid | customerid | amt
-----+-----+-----
10677   | 2           | 5.50
5019    | 3           | 277.44
9748    | 3           | 77.17
(3 rows)
```

Как это сделать

Во-первых, нет необходимости форсировать задачу. Проверках всех столбцов на уникальность может занять слишком много времени.

Начнем с собственного оптимизатора статистики PostgreSQL. Свежие статистические данные можно получить, запустив для таблицы команду

```
postgres=# analyze ord;  
ANALYZE
```

Она работает быстро, так что долго ждать не придется.

Теперь исследуем нужные нам столбцы

```
postgres=# SELECT attname, n_distinct  
           FROM pg_stats  
           WHERE schemaname = 'public'  
           AND tablename = 'ord';  
  
attname      | n_distinct  
-----+-----  
orderid      | -1  
customerid   | -0.666667  
amt          | -1  
(3 rows)
```

Этот пример был выбран потому, что в нем два возможных ответа. Если значение `n_distinct` равно `-1`, то столбец на основании представленных данных считается уникальным.

Теперь следует выяснить, являются ли эти столбцы (вместе или по отдельности) уникальными случайно или же так была спроектирована база данных, в которой они находятся.

Бывает, что ни один столбец не может уникально идентифицировать строки. В таком случае часто используют в качестве ключа несколько столбцов. Если нет ни одного уникального столбца, начинаем искать их комбинацию. Следующий запрос показывает плотность распределения таблицы: в одном случае значение повторяется дважды, в другом появляется только один раз:

```
postgres=# SELECT count as num_of_values, count(*)  
           FROM (SELECT customerid, count(*)  
                 FROM ord  
                 GROUP BY customerid) s  
           GROUP BY count  
           ORDER BY count(*);  
  
num_of_values | count  
-----+-----  
2             | 1  
1             | 1  
(2 rows)
```

Можно изменить запрос так, чтобы включить несколько столбцов:

```
SELECT count as num_of_values, count(*)  
FROM (SELECT column1, column2.... columnN  
      ,count(*)
```

```
FROM ord
GROUP BY column1, column2.... columnN
) s
GROUP BY count
ORDER BY count(*);
```

Если будет найдена уникальная комбинация столбцов, результатом выполнения запроса будет всего одна строка. Чем ближе вы подбираетесь к ключу, тем плотнее становится распределение.

Как это работает

Можно искать уникальный ключ с помощью программы, однако в большинстве случаев это гораздо быстрее сделает человек, пользуясь информацией об именах столбцов, внешних ключах или представлении о том, как все должно работать. Это значительно уменьшает время поиска, которое понадобилось бы при попытке решить задачу «в лоб».

`ANALIZE` берет фрагмент данных таблицы и проводит статистический анализ результатов. `n_distinct` определяет, сколько различных значений в столбце.

Генерирование тестовых данных

Администратору БД иногда требуются данные для тестов. Они нужны для запуска тестовой БД или же для проверки производительности SQL-кода.

Как это сделать

Чтобы создать тестовую таблицу, вам понадобятся строки, столбцы, порядок.

Строки

Для автоматического создания множества строк данных обычно используется функция, возвращающая выборку. Можно написать свою функцию или воспользоваться парой готовых, включенных в PostgreSQL.

Последовательность строк создается подобным запросом:

```
postgres=# SELECT * FROM generate_series(1,5);
generate_series
-----
1
2
```

```

3
4
5
(5 rows)

```

Или можно сформировать список дат:

```

postgres=# SELECT date(generate_series(now(), now() + '1
week', '1
day'));
      date
-----
2010-03-30
2010-03-31
2010-04-01
2010-04-02
2010-04-03
2010-04-04
2010-04-05
2010-04-06
(8 rows)

```

Любую из этих функций можно использовать, чтобы сгенерировать строки и соответствующий им первичный ключ.

Столбцы

Теперь требуется сгенерировать значения для каждого столбца тестовой таблицы. Эту задачу можно разбить на несколько функций, используя в качестве руководства следующий пример:

Случайные целые числа (integer):

```
(random()*(2*109))::integer
```

Случайные целые числа с большими значениями (bigint):

```
(random()*(9*1018))::bigint
```

Случайные числовые данные:

```
(random()*100)::numeric(4,2);
```

Последовательности (строки) случайной длины, но не более максимальной:

```
repeat('1',(random()*40)::integer)
```

Подпоследовательности случайной длины:

```
substr('abcdefghijklmnopqrstuvwxyz',1,
(random()*26)::integer)
```

Случайная строка из списка строк:

```
(ARRAY['one','two','three'])[1+random()*3]
```


Порядок

Если совместить первое и второе, получим таблицу:

```
postgres=# SELECT generate_series(1,10) as key
              ,(random()*100)::numeric(4,2)
              ,repeat('1',(random()*25)::integer);
```

key	numeric	repeat
1	83.05	1111
2	5.28	1111111111111111
3	41.85	1111111111111111111111
4	41.70	1111111111111111111
5	53.31	1
6	10.09	11111111111111111
7	68.08	111
8	19.42	1111111111111111
9	87.03	11111111111111111111
10	70.64	1111111111111111

(10 rows)

Или в случайном порядке:

```
postgres=# SELECT generate_series(1,10) as key
              ,(random()*100)::numeric(4,2)
              ,repeat('1',(random()*25)::integer)
              ORDER BY random();
```

key	numeric	repeat
6	70.31	1111111111111111111111
4	2.37	11111111111111111111
1	76.99	1111111111111111
8	35.90	111111111111111
3	59.21	111111111
2	88.86	11111111
7	67.32	111111
9	15.66	111111
5	79.90	11111
10	25.09	1

(10 rows)

Как это работает

Функции, возвращающие выборки, в буквальном смысле возвращают выборки строк. Это позволяет использовать их как в операторе `FROM` в качестве таблицы, так и в `SELECT`. Набор функций `generate_series()` может возвращать как целые числа, так и даты в зависимости от типов данных параметров, поданных на вход.

Оператор `::` используется для отделения различных типов данных.

В примере «случайная строка из списка строк» используются массивы PostgreSQL. Массив можно создать с помощью конструкции `ARRAY`, после чего использовать для ссылки на элементы массива целые числа. В данном случае использовались случайные индексы.

И еще

Для создания тестовых данных для приложений можно воспользоваться готовыми платными программами. Их цена не так уж велика, поэтому стоит упомянуть их в этом разделе:

- <http://www.sqlmanager.net/products/postgresql/datagenerator>;
- <http://www.datanamic.com/datagenerator/index.html>.

Главными преимуществами таких генераторов данных являются:

- возможность генерировать данные в правильном формате для собственных типов данных;
- возможность добавлять данные сразу в несколько таблиц с учетом связывающих их внешних ключей;
- возможность добавлять данные в неравномерные распределения.

А также

Все описанные программы работают хорошо, но существует одна проблема. В реальности среди данных могут попадаться такие странные случаи, которые трудно симулировать. Одной из наиболее сложных задач является генерирование данных, которые удовлетворяют реалистичному распределению. Например, если речь идет о росте людей, здесь подойдет нормальное распределение, для создания списка банковских счетов – распределение по Парето, а для числа страховых случаев, возможно (хотя и не обязательно), – распределение Пуассона. Попытка предусмотреть все варианты из реальной жизни может оказаться весьма трудоемкой.

Для генерирования тестовых баз данных можно воспользоваться фрагментом существующих данных. Как это сделать, вы узнаете из следующего раздела.

Случайные данные по примеру

Представьте, что вы администратор БД, которому поставили задачу настроить тестовый сервер и населить его тестовыми данными.

Сервер довольно старый, и места на диске маловато. Итак, вы задумываетесь о создании примера.

Цель работы состоит в уменьшении общего объема данных и ускорении последующего анализа. Некоторым так нравится подобная идея, что они даже не задумываются, какие данные подойдут для примера.

Как это сделать

Сначала следует осознать, что инструмента, который возьмет и отрежет кусочек от вашей БД, просто нет. Увы!.. И вы поймете почему, если прочитаете этот раздел.

Рассмотрим, какой SQL-код поможет вывести пример данных. На деле это довольно просто: здесь вам пригодится функция `random()` внутри оператора `WHERE`. Например:

```
postgres=# SELECT count(*) FROM mybigtable;
count
-----
10000
(1 row)

postgres=# SELECT count(*) FROM mybigtable
           WHERE random() < 0.01;
count
-----
    95
(1 row)

postgres=# SELECT count(*) FROM mybigtable
           WHERE random() < 0.01;
count
-----
   106
(1 row)
```

При этом `WHERE random() < 0.01` присваивает каждой строке случайное число в диапазоне от 0.0 до 1.0, а затем отбирает те строки, для которых это значение оказалось менее 0.01. Другими словами, для примера отобран 1 % строк таблицы. Вы легко можете модифицировать этот код и вывести необходимые для ваших нужд проценты.

Теперь вытащим данные для примера из базы данных. По некоторым причинам это хитрая задача. Во-первых, нет возможности задать `WHERE` для `pg_dump`. Во-вторых, если создать представление с помощью `WHERE`, все равно `pg_dump` возьмет только его определение, а не само представление.

Зато с помощью `pg_dump` можно сделать дамп всей базы данных, за исключением некоторого числа таблиц:

```
pg_dump --exclude-table=MyBigTable > db.dmp
pg_dump --table=MyBigTable -schema-only > mybigtable.schema
psql -c '\copy (SELECT * FROM MyBigTable
          WHERE random() < 0.01) to mybigtable.dat'
```

а затем подгрузить его в отдельную базу данных с помощью:

```
psql -f db.dmp
psql -f mybigtable.schema
psql -c '\copy mybigtable from mybigtable.dat'
```

Тем не менее лучше по возможности избегать создания таких примеров. Или хотя бы ограничиться несколькими большими таблицами. Так вы решите сразу две проблемы: выполните математическое моделирование примера и избежите сложностей с вытаскиванием данных.

Как это работает

Механизм извлечения демонстрирует возможности инструментов командной строки PostgreSQL `psql` и `pg_dump`. Функция `pg_dump` позволяет включать или исключать файлы и делать дампы всей таблицы или даже ее схемы, в то время как `psql` выполняет дампы результата произвольного запроса в файл.

Не будем останавливаться на том, насколько случайными являются результаты работы функции `random()`, поскольку не это является нашей целью. При желании вы можете выбрать другой внешний генератор случайных чисел и вызывать его в SQL с помощью функции языка C.

Приведенный метод является простым способом создания примеров модели *равновероятной выборки* (EPS – equal probability of selection).

Примеры EPS считаются полезными, поскольку вариации атрибутов выборки похожи на вариации первоначального набора данных. Однако имейте в виду, что польза здесь будет только относительно вариаций.

В полученный пример попадут, скорее всего, лишь самые часто повторяющиеся данные. Если ваша примерная выборка состоит из 1 % данных, в которых какой-либо вид данных появлялся только в 0.001 % случаев, то в примере его может не оказаться вообще.

Допускается предварительно разделить данные и взять различные примеры из каждой группы; тогда в полученной примерной выборке будет больше нестандартных атрибутов. Например, можно

- включить 1 % всех обычных данных;
- включить 25 % нестандартных данных.

Следует заметить, что в таком случае ваша модель уже не будет являться EPS.

А также

После прочтения этого раздела некоторых статистиков может хватить удар! Пожалуйста, используйте всю мощь языка SQL для создания более точной выборки. Убедитесь только, что вы знаете, что делаете, и/или обратитесь к хорошей литературе, интернет-ресурсам или учебникам по статистике.

Подгрузка данных из электронных таблиц

Электронные таблицы являются одним из самых распространенных хранилищ данных. Исследования показали, что более половины небольших массивов данных хранится в электронных таблицах или в БД на обычном компьютере. Поэтому подгрузка данных из подобных источников – явление довольно частое.

Подготовка

Электронные таблицы сочетают данные, презентацию и программы в одном файле. Это удобно пользователям, поскольку позволяет быстро решать задачи. PostgreSQL, как и другие реляционные СУБД, в основном работает с данным на самом низком уровне, поэтому извлечение именно данных может оказаться непростой задачей.

С данными из электронных таблиц легко работать, если они удовлетворяют определенным условиям:

- каждый столбец электронной таблицы является столбцом только одной таблицы;
- каждая строка электронной таблицы является строкой только одной таблицы;
- данные принадлежат только одной рабочей странице электронной таблицы;

- желательно, чтобы первая строка являлась списком определенных/заголовков столбцов.

Это очень простая модель, но, скорее всего, в электронной таблице будет присутствовать что-то еще: заголовки, комментарии, константы для формул, итоговые строки, макросы, иллюстрации и т. д. В таком случае лучше всего создать новую рабочую страницу электронной таблицы, удовлетворяющую перечисленным условиям, а затем поставить перекрестные ссылки, чтобы получить данные. Пример такой ссылки: `"=Sheet2.A1"`. Необходимо выделить отдельную рабочую страницу для каждой выборки данных, которая в PostgreSQL станет таблицей. Хотя допускается в одну таблицу подгружать несколько рабочих страниц.

Некоторые пользователи электронных таблиц могут посчитать все это бесполезной тратой времени и доказательством неудобства БД. Специалисты по электронным таблицам выступают за такой формат: все данные на одной рабочей странице, вычисления и презентация – на других. Действительно, такая модель идеальна, однако работать-то нам приходится в реальном мире.

Как это сделать

Если данные в электронной таблице аккуратно сложены на одной рабочей странице, как показано на иллюстрации, вы можете просто выполнить команды **File** → **Save As**, указав расширение **CSV**.

Chapter5_Sample_Spreadsheet.csv - OpenOffice.org Calc								
File	Edit	View	Insert	Format	Tools	Data	Window	Help
	A	B	C	D				
1	Key	Value						
2		1c						
3		2d						
4								

Рис. 5.1

Текущая рабочая страница будет экспортирована в подобный файл:

```
"Key", "Value"  
1, "c"  
2, "d"
```

После чего его можно подгрузить в уже существующую таблицу PostgreSQL с помощью команды `psql`:

```
postgres=# \COPY sample FROM sample.csv CSV HEADER
postgres=# SELECT * FROM sample;
 key | value
-----+-----
   1 | c
   2 | d
```

Или из командной строки:

```
psql -c '\COPY sample FROM sample.csv CSV HEADER'
```

Если данные находятся в другой директории, файл будет включать полный путь. Команда `psql \COPY` передает данные из клиентской системы, в которой она была запущена, на сервер БД, так что файл остается у клиента.

Если вы выполняете SQL по соединению другого типа, можно использовать следующее SQL-выражение:

```
COPY sample FROM '/mydatafiledirectory/sample.csv' CSV HEADER;
```

Этот оператор выполняется на сервере БД и может быть запущен только суперпользователем. Таким образом, вам нужно будет самостоятельно перенести данные на сервер, а затем подгрузить их. Команда `COPY` в представленном SQL-выражении использует для нахождения файлов с данными абсолютный путь.

Команда `COPY` (или `\COPY`) не создает таблицу – это необходимо сделать заранее. Следует отметить, что опция `HEADER` ничего не делает и нужна лишь для того, чтобы проигнорировать первую строку входного файла, так что имена столбцов из файла `.csv` не обязательно должны соответствовать таблице в Postgres. Если в файле отсутствует строка с заголовками, то `HEADER` проигнорирует первую строку с данными. К сожалению, PostgreSQL не может отличить строку с данными от строки с заголовками. Будьте внимательны!

Стандартного инструмента для подгрузки данных непосредственно из электронной таблицы в базу данных нет. Для автоматизации перечисленных задач нетрудно написать макрос для электронной таблицы, однако рассмотрение этой темы выходит за рамки нашей книги.

Как это работает

Команда `\COPY` выполняет SQL-выражение `COPY`, так что два представленных метода очень похожи. О команде `COPY` можно сказать кое-что еще, и мы поговорим об этом в следующем разделе.

А также

Есть много инструментов для извлечения и подгрузки данных. Есть дорогие, есть и дешевые. Однако следует помнить, что самой сложной задачей при подгрузке данных из электронных таблиц является отделение собственно данных от всего остального. Авторам книги еще не встречались инструменты, которые могли бы в этом помочь.

Подгрузка данных из файлов

Подгрузка данных в базу данных – одна из самых важных задач. Делать это нужно быстро и аккуратно. Итак, приступим.

Подготовка

Вам понадобится утилита `pgloader`, которую можно найти по адресу <http://pgloader.projects.postgresql.org/>.

Как это сделать

В PostgreSQL есть команда `copy`, которая предназначена для подгрузки/выгрузки, однако подгрузку данных она выполняет не слишком хорошо. Оставим ее в покое и перейдем непосредственно к утилите `pgloader`.

Чтобы подгружать данные, необходимо определиться с требованиями. Опишем этот процесс поэтапно:

1. Определите файлы с данными и выясните, где они находятся. Убедитесь что программа `pgloader` установлена там, где находятся файлы.
2. Найдите таблицу, в которую нужно подгружать файлы, убедитесь в том, что у вас есть соответствующие права и достаточно места.
3. Выясните тип файла (фиксированный, текстовый или CSV) и проверьте кодировку.
4. Определите связь между столбцами в файле и столбцами таблицы, в которую будет производиться подгрузка. Подумайте над тем, какие столбцы из файла не понадобятся – `pgloader` позволяет включать только необходимые столбцы. Определите, в каких столбцах таблицы нет данных. Нужно ли оставить там значения по умолчанию или `pgloader` должна сгенерировать для этих столбцов значения с помощью функций или констант?

5. Задайте любые преобразования, которые нужно выполнить при загрузке. Обычно это необходимо для формата записи дат, но бывают и другие случаи.
6. Напишите скрипт для `pgloader` и протестируйте его с помощью `--dry-run`. Затем попытайтесь загрузить несколько записей с помощью `--count`.
7. Решите, нужен ли вам журнал для фиксирования произошедших или сорванных загрузок и потребуется ли для этого автоматизация. Кроме того, определите, что делать со строками в случае отказа и где хранить журнал с записями отклоненных попыток, если он переполнится.
8. Наконец, подумайте о производительности. Этот пункт стоит на последнем месте, и если он вступает в противоречие с каким-либо другим, то на первом месте должна быть корректная работа по загрузке данных.

Да, для выполнения программы `pgloader` рекомендуется использовать скрипт. Так проще повторять ее запуск, поскольку загрузить данные с первого раза получается разве что в кино.

Рассмотрим типичный пример – `cookbook_pgloader.conf`:

```
[pgsql]
host = 192.168.0.5
base = pgloader
user = dim
log_file           = /tmp/pgloader.log
log_min_messages   = DEBUG
client_min_messages = WARNING
lc_messages        = C
client_encoding    = 'utf-8'
copy_every         = 10000
null = ''
empty_string       = '\ '
max_parallel_sections = 4
[load]
table              = load_example
filename           = simple/simple.data
format             = csv
datestyle          = dmy
field_sep          = |
trailing_sep       = True
columns            = a:1, b:3, c:2
reject_log         = /tmp/simple.rej.log
reject_data        = /tmp/simple.rej
section_threads    = 4
```

Можно воспользоваться скриптом загрузки:

```
pgloader --summary --vacuum --config cookbook_pgloader.conf
```

Как это работает

Утилита `pgloader` успешно справляется с ошибками. Функция `copy` подгружает все строки одной транзакцией, поэтому единственная ошибка может оборвать всю подгрузку. `pgloader` разбивает исходный файл на относительно небольшие куски и подгружает их постепенно. Если какие-то строки приводят к ошибкам, `pgloader` постепенно пропускает их и отсеивает «плохие».

Утилита `pgloader` написана на языке Python и позволяет соединяться с PostgreSQL из стандартного клиентского интерфейса Python. Да, эффективность у `pgloader` ниже, чем у `copy`, но для работы `copy` есть много ограничений: файл правильного формата должен лежать на сервере в определенном месте и не содержать ошибок. Утилита `pgloader` требует больших ресурсов, но зато она подгружает данные параллельными потоками и поэтому работает быстрее. Кроме того, во многих случаях `pgloader` может переформатировать функции, написанные на Python, а команда `copy` слишком простая.

Утилита `pgloader`, в отличие от `copy`, позволяет также подгружать данные из файлов с фиксированной шириной.

И еще

Если вам необходимо повторно подгрузить данные, чтобы обновить их, в командной строке `pgloader` используйте `--truncate`.

Если при подгрузке возникали ошибки, в таблицы PostgreSQL мог попасть мусор. Не тот мусор, который можно увидеть и который вызывает смысловые ошибки, а скорее фрагментация. Чтобы избежать ее, используйте дополнительную опцию `--vacuum`, однако в этом случае подгрузка может занять гораздо больше времени.

Необходимо также убедиться, что одни и те же данные не будут подгружены дважды: проверьте индексы. У каждой подгружаемой таблицы должен быть один уникальный индекс. В противном случае подгрузка быстро оборвется.

Очень часто проблемы вызывает обработка строковых данных, поскольку в них могут присутствовать непечатаемые символы. По умолчанию в PostgreSQL отключен параметр `standard_conforming_strings`: знак `\` (обратная косая черта) воспринимается как escape-символ. Другими словами, по умолчанию `'\n'` означает разрыв строки, что может привести к отсеканию части данных. Необходимо включить этот параметр (`standard_conforming_strings = on`) или определить escape-символы в файле с параметрами загрузки.

Если вы хотите переподгрузить данные, которые были выгружены из PostgreSQL, возможно, лучше использовать утилиту `pg_restore`. У нее есть опция переподгрузки данных в параллельных `-j`-потоках, хотя она доступна только в том случае, если дамп был выполнен с использованием формата `custom pg_dump`. Подробнее об этом можно прочитать в одиннадцатой главе. С `pg_restore` удобно переподгружать дампы, но преимуществ `pgloader`, описанных здесь, там не хватает.

А также

Вы можете написать Димитрию Фонтейну, автору и разработчику `pgloader`. Он любит получать письма от пользователей.



ГЛАВА 6.

Безопасность

Рассматриваемые темы:

- Как забрать у пользователя права доступа к таблице
- Как предоставить пользователю доступ к таблице
- Создание нового пользователя
- Временный запрет соединений для пользователя
- Удаление пользователя без потери его данных
- Проверка безопасности паролей пользователей
- Назначение определенным пользователям ограниченных прав суперпользователя
- Аудит изменений DDL
- Аудит изменений данных
- Интеграция с LDAP
- Соединение с использованием SSL
- Шифрование секретных данных

Введение

БД обычно используют для хранения различных данных, что налагает на использование баз некоторые ограничения. Некоторые записи или таблицы могут быть видны только некоторым пользователям, и даже к таблицам, видимым всем, есть разные уровни допуска: не всем разрешено добавлять новые данные или изменять уже существующие. У каждого пользователя есть определенные права доступа к таблицам и другим объектам БД, таким как схемы или функции.

Предпочтительнее не присваивать роли непосредственно пользователям, а использовать промежуточную роль (ROLE), для которой назначаются права. Затем она присваивается подходящим пользователям. Например, роль `clerk` может иметь права для вставки и обновления данных в таблице `user_account`, но только права для вставки данных в таблицу `audit_log`.

С другой стороны, требуется, чтобы только определенные люди могли иметь доступ к БД, чтобы пользователь не мог видеть, что

делают в БД остальные (если, конечно, он не администратор или аудитор), чтобы пользователи не могли передавать права, которые были им даны.

Следует учитывать и еще один немаловажный аспект: сервер должен находиться в безопасном месте, и процедуры доступа к нему должны быть безопасными. Однако вопрос общей защиты БД, серверной машины и сети настолько обширен, что его сложно осветить в этой книге.

Если вас беспокоит данная проблема, читайте книги и статьи, посвященные защите, или наймите консультанта по безопасности. Защита БД – всего лишь капля в море задач, связанных с безопасностью.

Обычная роль пользователя

На работающем сервере все пользователи делятся как минимум на две группы: администраторы и конечные пользователи. Причем администраторы могут делать все (являются суперпользователями), а конечные пользователи могут совсем немного: как правило, изменять данные в нескольких таблицах и читать еще несколько таблиц.

Не очень разумно давать простым пользователям право создавать или изменять определения объектов БД, а значит, они не должны иметь права `CREATE` для всех схем, включая `PUBLIC`.

Для конечных пользователей существуют и другие роли. Аналитики, например, могут только делать выборку данных из одной таблицы или представления или выполнять несколько функций. А менеджер уполномочен только давать или отнимать права, но больше ничего делать не должен.

Как забрать у пользователя права доступа к таблице

Материал данного раздела дает ответ на вопрос «Как запретить пользователю X доступ к таблице Y?».

Подготовка

Текущий пользователь должен быть суперпользователем, владельцем таблицы или иметь доступ `GRANT` для этой таблицы. Вы можете отнять права и у пользователя, который является суперпользователем.

Как это сделать

Чтобы отнять все права на таблицу `mysecrettable` у пользователя `userwhoshouldnotseeit`, необходимо выполнить следующую SQL-команду:

```
REVOKE ALL ON mysecrettable FROM userwhoshouldnotseeit;
```

Однако таблица все еще остается открытой для пользователей через роль `PUBLIC`, поэтому следует также записать:

```
REVOKE ALL ON mysecrettable FROM PUBLIC;
```

Как это работает

По умолчанию у всех пользователей есть права (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES` и `TRIGGER`) на все вновь созданные таблицы посредством специальной роли `PUBLIC`. Чтобы определенный пользователь больше не мог получить доступ к таблице, права на нее необходимо отнять и у этого пользователя, и у роли `PUBLIC`.

И еще

Опция `GRANT` и `CASCADE`.

```
REVOKE ... CASCADE
```

Лучше всего

На рабочих системах лучше всего всегда включать выражения `GRANT` и `REVOKE` в скрипт создания БД, и тогда вы будете уверены, что доступ к этой базе получают только нужные пользователи. Если делать это вручную, можно что-нибудь забыть. Кроме того, таким образом вы удостоверитесь, что одни и те же роли используются для разработки и тестирования окружения, так что на стадии развертывания с этой стороны неожиданностей не будет.

Вот пример части скрипта для создания базы данных:

```
CREATE TABLE table1(  
...  
);  
REVOKE ALL ON table1 FROM GROUP PUBLIC;  
GRANT SELECT ON table1 TO GROUP webreaders;  
GRANT SELECT, INSERT, UPDATE, DELETE ON table1 TO editors;  
GRANT ALL ON table1 TO admins;
```

Путь для поиска по умолчанию

При отъеме или назначении прав рекомендуется использовать полное имя, иначе может неожиданно оказаться, что вы работали не с той таблицей.

Чтобы посмотреть эффективный путь БД, выполните:

```
pguser=# show search_path ;
search_path
-----
*$user*,public
(1 row)
```

Чтобы увидеть, на какую таблицу будет оказано влияние, если вы опустите имя схемы, запустите в `psql`:

```
pguser=# \d x
Table "public.x"
Column | Type | Modifiers
-----+-----+-----
```

В результате будет получено полное имя таблицы `public.x`, включая схему.

Как предоставить пользователю доступ к таблице

Чтобы выполнять какие-либо действия над таблицей, пользователь должен иметь права доступа к ней. По умолчанию PostgreSQL дает всем полные права посредством роли `PUBLIC`, но, если настройки были изменены для усиления безопасности, при создании таблицы права на нее у роли `PUBLIC` могут быть отозваны.

Подготовка

Убедитесь, что у вас определены соответствующие роли и что права у роли `PUBLIC` отозваны.

Как это сделать

Предоставьте доступ к схеме, содержащей таблицу:

```
GRANT ALL ON someschema TO somerole;
GRANT SELECT, INSERT, UPDATE, DELETE ON someschema.
sometable TO
somegroup;
GRANT somerole TO someuser, otheruser;
```

Как это работает

Указанные выше команды сначала дают роли полный доступ к схеме, права на просмотр (`SELECT`) и изменение (`INSERT`, `UPDATE`, `DELETE`), а затем назначают эту роль двум пользователям БД.

И еще

В PostgreSQL нет требования одних прав для получения других: у пользователя могут быть таблицы только для записи, куда нужно вставлять данные, но нельзя их просматривать. Администратор обычно создает по этому принципу очередь сообщений: сотрудники отправляют данные одному из менеджеров, но не видят, что отправили другие.

С другой стороны, вы не сможете изменить или удалить сделанную вами запись. Это необходимо для аудита таблиц журнала, в котором фиксируются все изменения и который нельзя модифицировать.

Также нужен доступ к схеме

Чтобы получить доступ к таблице, пользователь должен иметь доступ к схеме, содержащей эту таблицу:

```
GRANT USAGE ON SCHEMA someschema TO someuser;
```

Как дать доступ к таблице через групповую роль

Часто приходится давать группам пользователей схожие права на какие-либо объекты БД. Для этого сначала необходимо предоставить все права промежуточной роли (или разрешенной группе), а затем принять в эту группу избранных пользователей:

```
CREATE GROUP webreaders;  
GRANT SELECT ON pages TO webreaders;  
GRANT INSERT ON viewlog TO webreaders;  
GRANT webreaders TO tim, bob;
```

Теперь, и `tim`, и `bob` имеют права `SELECT` на таблицу `pages` и `INSERT` на таблицу `viewlog`. Можно также добавлять права роли уже после принятия в нее пользователей. Таким образом, после

```
GRANT INSERT, UPDATE, DELETE ON comments TO webreaders;
```

у пользователей `bob` и `tim` появятся все эти права на таблицу `comments`.

Как дать доступ ко всем объектам в схеме

В более ранних версиях PostgreSQL не было простого способа предоставить права для работы с двумя или несколькими объектами, кроме перечисления их в команде `GRANT` и `REVOKE`.

В версии 9.0 команды `GRANT` или `REVOKE` можно распространять на все однотипные объекты схемы:

```
GRANT SELECT ON ALL TABLES IN SCHEMA staging TO bob;
```


Но все еще нужно дополнительно отдельной командой давать права доступа к схеме.

Создание нового пользователя

В этом рецепте рассматриваются два способа создания нового пользователя БД из командной строки и один способ – с помощью команд SQL.

Подготовка

Чтобы создавать новых пользователей, вы должны быть суперпользователем или иметь права `createrole` или `createuser`.

Как это сделать

В командной строке запустите команду `createuser` и ответьте на несколько вопросов:

```
pguser@hvost:~$ createuser bob
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) y
Shall the new role be allowed to create more new roles?
(y/n) n
pguser@hvost:~$ createuser tim
Shall the new role be a superuser? (y/n) y
```

Как это работает

Программа `createuser` – это просто обертка для выполнения команд SQL над кластером БД. Она подключается к базе данных `postgres`, задает вопрос, а затем выполняет команды SQL для создания пользователя. То же самое можно сделать, запустив SQL-команду `CREATE USER`:

```
CREATE ROLE bob WITH NOSUPERUSER INHERIT NOCREATEROLE
CREATEDB LOGIN;
CREATE ROLE tim WITH SUPERUSER;
```

И еще

Проверка ролей пользователя:

```
pguser=# \du tim
               List of roles
Role name | Attributes | Member of
-----+-----+-----
Tim      | Superuser  | {}
```

```

      : Create role
      : Create DB
pguser=# \du bob
      List of roles
Role name | Attributes | Member of
-----+-----+-----
bob      | Create DB | {}

```

CREATE USER и CREATE GROUP

Начиная с версий 8.x, команды `CREATE USER` и `CREATE GROUP` фактически являются вариациями команды `CREATE ROLE`.

Выражение `CREATE USER u`; эквивалентно `CREATE ROLE u LOGIN`; а выражение `CREATE GROUP g`; равносильно `CREATE ROLE g NOLOGIN`;

Временный запрет соединений для пользователя

Иногда необходимо временно отозвать у пользователя права на подключение, при этом не удаляя пользователя и не меняя его пароль.

В следующем разделе будет показано, как это можно сделать.

Подготовка

Чтобы менять права доступа пользователей, вы должны быть супер-пользователем или иметь право `REVOKE` (в этом случае вы не сможете менять права суперпользователей).

Как это сделать

Временно запретить пользователю вход в систему можно так:

```

pguser=# alter user bob nologin;
ALTER ROLE

```

Чтобы вернуть пользователю возможность устанавливать соединение, выполните

```

pguser=# alter user bob login;
ALTER ROLE

```

Как это работает

В системном каталоге PostgreSQL ставится флажок, запрещающий пользователю вход в систему. При этом текущие соединения не разрываются.

Ограничение числа одновременных соединений пользователя

Есть и другой способ не дать пользователю войти в систему. Вы можете установить число возможных соединений для этого пользователя (connection limit) равным 0:

```
pguser=# alter user bob connection limit 0;
ALTER ROLE
```

Если вы хотите разрешить пользователю bob устанавливать до 10 соединений одновременно, выполните:

```
pguser=# alter user bob connection limit 10;
ALTER ROLE
```

Кроме того, можно полностью снять ограничение числа возможных соединений:

```
pguser=# alter user bob connection limit -1;
ALTER ROLE
```

Отключение пользователей, имеющих свойство NOLOGIN

Если вы отозвали права на установление соединений у пользователей и хотите сейчас же отключить их, выполните следующее (для этого у вас должны быть права суперпользователя):

```
SELECT pg_terminate_backend(procpid)
FROM pg_stat_activity a
JOIN pg_roles r ON a.username = r.rolname AND not rolcanlogin;
```

В ранних версиях PostgreSQL, где нет функции pg_terminate_backend(), вы можете в командной строке на сервере набрать в качестве пользователя postgres:

```
postgres@hqvost:~$ psql -t -c "\
select 'kill ' || procpid from pg_stat_activity a \
join pg_roles r on a.username = r.rolname and not
rolcanlogin;" \
| bash
```

и получить тот же результат.

В этом случае из запроса формируются команды kill, которые направляются затем оболочке для выполнения.

Удаление пользователя без потери его данных

При попытке сбросить пользователя, работающего с таблицами или другими объектами БД, вы получите следующее сообщение об ошибке:

```
testdb=# drop user bob;  
ERROR: role "bob" cannot be dropped because some objects  
depend on it  
DETAIL: owner of table bobstable  
owner of sequence bobstable_id_seq
```

Решение проблемы вы узнаете, прочитав нижеследующий раздел.

Подготовка

У вас должны быть права суперпользователя или право `CREATEROLE`.

Как это сделать

Проще всего не сбрасывать пользователя, а воспользоваться трюком из предыдущего раздела, запретив пользователю устанавливать соединения:

```
pguser=# alter user bob nologin;  
ALTER ROLE
```

Еще один плюс состоит в том, что при последующем аудите и тестировании вы будете знать, кто создал таблицу или откуда она взялась.

Как это работает

Пользователю назначается роль без права входа в систему.

Как передать права пользователя, которого вы удалили, новому пользователю

```
pguser=# grant bob bobs_replacement;  
GRANT
```

Как сделать владельцами объектов других пользователей

Если вам действительно нужно избавиться от пользователя, вы должны передать все, чем он владел, другому пользователю, так что выполните следующий запрос, который является дополнением PostgreSQL к стандарту SQL:

```
REASSIGN OWNED BY bob TO bobs_replacement;
```

При этом владение всеми объектами БД, которые принадлежали роли `bob`, будет передано роли `bobs_replacement`.

Однако для этого у вас должны быть права изменения обеих ролей, и необходимо повторить этот запрос во всех базах данных, где

bob владеет какими-либо объектами, поскольку REASSIGN OWNED работает только в текущей базе данных.

Переназначение владельца для старых баз данных

Команда REASSIGN OWNED была добавлена в PostgreSQL в версии 8.2. Если ваша БД имеет более раннюю версию, вам придется поработать в командной строке Unix.

Для начала извлеките из дампа схемы назначения прав:

```
dbuser:~$ pg_dump -s mydatabase | grep -i "alter.*owner to bob"
ALTER FUNCTION public.somefunction() OWNER TO bob;
ALTER TABLE public.directory OWNER TO bob;
ALTER TABLE public.directory_seq OWNER TO bob;
ALTER TABLE public.document_id_seq OWNER TO bob;
ALTER TABLE public.documents OWNER TO bob;
```

Затем просто замените в полученном результате bob на нового пользователя и передайте команды обратно базе данных:

```
dbuser:~$ pg_dump -s mydb | grep -i "owner to bob" > tmp.sql
sql
dbuser:~$ sed -e 's/TO bob/TO bobs_replacement/' < tmp.sql
| psql
mydb
```

И хорошо бы посмотреть сначала на измененные данные:

```
dbuser:~$ pg_dump -s mydb | grep -i "owner to bob" > tmp.sql
dbuser:~$ sed -e 's/TO bob/TO bobs_replacement/' < tmp.sql
>tmp2.sql
dbuser:~$ less tmp2.sql
dbuser:~$ psql mydb < tmp2.sql
```

Как и в случае с командой REASSIGN OWNED, это работает каждый раз только для одной базы данных, так что следует все повторить для каждой БД, если в ней находятся объекты, принадлежащие роли, которую вы хотите удалить.

Проверка безопасности паролей пользователей

В PostgreSQL нет встроенной возможности проверки паролей.

Максимум, что можно сделать, — убедиться, что все пароли пользователей зашифрованы и что файл pg_hba не позволяет подключаться

с обычным паролем. То есть в качестве метода аутентификации пользователей всегда используйте MD5.

Для подключения клиентских приложений из доверенных частных сетей, реальных или виртуальных (VPN), можно использовать доступ по узлу (host based access) – если вы уверены, что машину, на которой запущено приложение, не используют подозрительные личности. Для удаленного доступа из публичных сетей лучше применять клиентские сертификаты SSL.

Подготовка

Узнать, у каких пользователей незашифрованные пароли, можно с помощью запроса:

```
test2=# select username,password from pg_shadow where password
not like
'md5%' or length(password) <> 35;
username | password
-----+-----
tim      | weakpassword
asterisk | md5chicken
(2 rows)
```

Пользователей с зашифрованными паролями показывает следующий запрос:

```
test2=# select username,password from pg_shadow where password
like 'md5%'
and length(password) = 35;
username | password
-----+-----
bob2     | md518cf038878cd04fa207e7f5602013a36
(1 row)
```

Как это сделать

Шифрование паролей БД – это еще только половина дела.

Гораздо хуже то, что неизвестно, действительно ли пользователи выбирают пароли, которые трудно угадать. Например, «password», «secret» или «test», да и любые общеупотребительные слова, являются не очень хорошими паролями.

Если вы не верите, что пользователь может придумать хороший пароль, напишите приложение-оболочку, которое будет проверять качество пароля, и направлять его на пользователей, когда они меняют пароль. На странице <http://developer.postgresql.org/pgdocs/postgres/passwordcheck.html> вы найдете модуль для решения этой задачи в некоторых случаях (если пароль приходит от клиента в виде простого текста).

Назначение определенным пользователям ограниченных прав суперпользователя

У роли суперпользователя есть такие права, которые можно назначить и другим ролям, не являющимся суперпользователями.

Например, роли `bob` предоставляется возможность создавать новые базы данных:

```
ALTER ROLE bob WITH CREATEDB;
```

Для создания новых пользователей понадобится назначить право `CREATEUSER`:

```
ALTER ROLE bob WITH CREATEUSER;
```

Однако обычным пользователям можно давать и более узкие права и контролируемый доступ к некоторым действиям, зарезервированным для суперпользователей, с помощью функций `SECURITY DEFINER`. Аналогично можно передавать часть прав одного пользователя другому.

Подготовка

Для начала вам необходимо иметь права суперпользователя БД. Предположим, что вы – суперпользователь по умолчанию `postgres`.

Продemonстрируем два случая назначения обычному пользователю прав, доступных только суперпользователям.

БД должна поддерживать внутренний язык `PL/pgSQL`. Начиная с версии 9.0 рекомендуется по умолчанию устанавливать поддержку `PL/pgSQL` во все новые базы данных, однако эту опцию могли изменить создатели пакетов или администраторы. Если поддержки нет, как суперпользователь `PostgreSQL` выполните следующее:

```
test2=# CREATE LANGUAGE plpgsql;  
CREATE LANGUAGE
```

Как это сделать

Одна из возможностей суперпользователей, которой нет у обычных пользователей, – дать указание `postgres` копировать табличные данные из файла:

```
pguser@hqvost:~$ psql -U postgres  
test2  
...
```

```
test2=# create table lines(line text);
CREATE TABLE
test2=# copy lines from '/home/bob/names.txt';
COPY 37
test2=# SET SESSION AUTHORIZATION bob;
SET
test2=> copy lines from '/home/bob/names.txt';
ERROR: must be superuser to COPY to or from a file
HINT: Anyone can COPY to stdout or from stdin. psql's \copy
command
also works for anyone.
```

Чтобы пользователь `bob` мог копировать данные непосредственно из файла, суперпользователь может написать для него специальную функцию-обертку:

```
create or replace function copy_from(tablename text,
filepath text)
returns void
security definer
as
$$
declare
begin
    execute 'copy ' || tablename || ' from ''' ||
filepath || ''';
end;
$$ language plpgsql;
```

И хорошо бы сделать так, чтобы эту функцию мог применять только определенный пользователь:

```
revoke all on function copy_from( text, text) from public;
grant execute on function copy_from( text, text) to bob;
```

Кроме того, можно проверить, чтобы `bob` копировал файлы только из своей домашней директории.

Как это работает

Когда вызывается функция с `security definer`, Postgres меняет права сессий для пользователя, который определил ее при выполнении.

Таким образом, когда `bob` вызывает функцию `copy_from(tablename, filepath)`, он фактически на время ее выполнения становится суперпользователем.

Это похоже на функцию `setuid` в Unix-системах, где можно сделать так, чтобы все, у кого есть право исполнения программы, запускали ее на правах ее владельца. Риски при этом тоже сходные.

И еще

Есть и другие операции, зарезервированные только для суперпользователей PostgreSQL, например настройка некоторых параметров.

Написание функции `debugging_info` для разработчиков

Некоторые параметры, контролирующие запись в журнал, могут определяться только суперпользователями.

Если вы хотите дать своим разработчикам право настраивать добавление записей в журнал, можно написать такую функцию:

```
create or replace function debugging_info_on()
returns void
security definer
as
$$
begin
    set client_min_messages to 'DEBUG1';
    set log_min_messages to 'DEBUG1';
    set log_error_verbosity to 'VERBOSE';
    set log_min_duration_statement to 0;
end;
$$ language plpgsql;
revoke all on function debugging_info_on() from public;
grant execute on function debugging_info_on() to bob;
```

После этого можно вернуться к первоначальному варианту, присвоив всем переменным значения по умолчанию:

```
create or replace function debugging_info_reset()
returns void
security definer
as
$$
begin
    set client_min_messages to DEFAULT;
    set log_min_messages to DEFAULT;
    set log_error_verbosity to DEFAULT;
    set log_min_duration_statement to DEFAULT;
end;
$$ language plpgsql;
```

В данном случае нет смысла возиться с `GRANT` и `REVOKE`, поскольку в возвращении значений по умолчанию нет никакого риска. Вместо `SET xxx to DEFAULT` можно использовать более короткую версию `RESET xxx`.

Или просто завершите сессию, так как эти параметры действуют только для текущей сессии.

Аудит изменений DDL

Ниже описывается, как собрать из журналов БД DDL (Data Definition Language), чтобы отследить изменения структуры базы данных.

Подготовка

В файле `postgresql.conf` укажите:

```
log_statement = 'ddl'
```

Можно также указать значение `mod` или `all`. И не забудьте переподгрузить новую конфигурацию:

```
/etc/init-d/postgresql reload
```

Как это сделать

Найдите в журнале все упоминания команд `CREATE`, `ALTER` и `DROP`.

```
postgres@hqvost:~$ egrep -i "create|alter|drop" \ /var/log/  
postgresql/  
postgresql-8.4-main.log
```

Если имеется ротация журналов, то необходимо также применить `grep` (быстрый просмотр) для всех журналов.

Если журнал слишком старый и вы не сохранили предыдущие журналы где-то в другом месте, то вам не повезло.

По умолчанию в `postgresql.conf` по поводу ротации указано:

```
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'  
log_rotation_age = 1d  
log_rotation_size = 10MB
```

Проверьте это, если думаете, что вам могут понадобиться журналы более чем недельной давности.

Как это работает

В разделе «Подготовка» вы дали команду PostgreSQL записывать все команды DDL в основной журнал.

В разделе «Как это сделать» описано, как вытащить DDL-запросы из файла журнала.

И еще

Были ли зафиксированы изменения

Вполне возможно, что некоторые изменения отражены в журнале, но их не видно в структуре базы данных. Большую часть DDL-команд

PostgreSQL можно «откатить» (`ROLLBACK`), так что в журнале окажется просто список команд, выполненных PostgreSQL, но не зафиксированных. Журнал зависит не от транзакций, поэтому в нем хранятся и команды, к которым применили `ROLLBACK`.

Кто совершил изменения

Чтобы знать, какой пользователь совершил DDL-изменения, необходимо их также записывать в журнал.

Это можно обеспечить, включив в параметр `log_line_prefix` строку в формате `%u`. Минимальный рекомендованный формат `log_line_prefix` для отслеживания DDL — `%t %u %d`; при этом каждая строчка журнала будет начинаться с указания времени, пользователя и базы данных.

Как получить эту информацию из базы данных

Если ведение журнала отключено или у вас есть не все журналы, то некоторую часть информации о том, кто и когда изменял схему БД, можно получить из системных таблиц, хотя это не самые надежные сведения.

Что вы действительно узнаете, так это владельца объекта БД (таблицы, последовательности, функции и т. п.), но владелец мог быть изменен с помощью `ALTER TABLE ... SET OWNER to yyyy`.

По идентификатору транзакции в столбце `xmin` в системных таблицах `pg_class` и `pg_attr` можно определить примерное время создания объекта или время последнего изменения. Попробуйте найти близкие `xmin` в какой-нибудь таблице, в которой автоматически записывается время вставки; возможно, в каком-либо столбце определено `DEFAULT CURRENT_TIMESTAMP`.

Аудит изменений данных

Здесь собраны различные способы получения информации об изменении данных, содержащихся в таблицах, для проведения аудита.

Подготовка

Сначала вам нужно решить некоторые вопросы:

- Нужно отслеживать все изменения или только некоторые?
- Какая информация об изменениях вам нужна? Возможно, только факт изменений?

- Если есть новое значение поля или кортежа, нужно ли знать старое?
- Достаточно ли записать пользователя, который произвел изменения, или необходима информация о его IP-адресе и других настройках соединения?
- Насколько безопасной (защищенной) должна быть информация о проведенном аудите? Например, есть ли необходимость хранить ее отдельно от исследуемой базы данных?

В зависимости от ответов можно выбрать метод, который подходит именно вам.

Как это сделать

Чтобы получить информацию об изменении данных из журнала сервера, выполните следующее:

- укажите `log_statement = 'mod' или 'all'`;
- выберите в журнале все команды `INSERT`, `UPDATE`, `DELETE` и `TRUNCATE`;
- или укажите способ хранения журналов на сервере БД или копирования на сторонний узел.

Чтобы собрать информацию с помощью триггеров, выполните следующие шаги:

1. Напишите функцию-триггер для сбора новых (а если нужно, то и старых) значений кортежей и сохранения их в специальных таблицах для аудита.
2. Добавьте эти триггеры к таблицам, которые необходимо отслеживать.

Ниже приведен несколько измененный пример выполнения функции A PL/pgSQL Trigger Procedure For Auditing из документации по PostgreSQL.

```
CREATE TABLE emp (  
    empname      text      NOT NULL,  
    salary       integer  
);  
  
CREATE TABLE emp_audit(  
    operation    text      NOT NULL,  
    stamp        timestamp NOT NULL,  
    userid       text      NOT NULL,  
    empname      text      NOT NULL,  
    salary       integer  
);
```

```
CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS
TRIGGER AS
$emp_audit$
BEGIN
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO emp_audit SELECT 'DEL', now(), user,
OLD.*;
    ELSIF (TG_OP = 'UPDATE') THEN
        -- save old and new values
        INSERT INTO emp_audit SELECT 'OLD', now(), user,
OLD.*;
        INSERT INTO emp_audit SELECT 'NEW', now(), user,
NEW.*;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO emp_audit SELECT 'INS', now(), user,
NEW.*;
    ELSEIF (TG_OP = 'TRUNCATE') THEN
        INSERT INTO emp_audit SELECT 'TRUNCATE', now(),
user,
'- ', -1;
    END IF;
    RETURN NULL; -- result is ignored because this is
an AFTER
trigger
END;
$emp_audit$ LANGUAGE plpgsql;
CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
CREATE TRIGGER emp_audit_truncate
AFTER TRUNCATE ON emp
FOR EACH STATEMENT EXECUTE PROCEDURE process_emp_
audit();
```

Сбор изменений с помощью триггеров и сохранение их в другой базе данных посредством dblink или plproxy

Для некоторых систем, требующих особой защиты, не подходит хранение журнала аудита на одной машине со всеми основными данными. В таком случае можно настроить удаленную запись изменений. Это можно сделать единственным способом: с помощью `pl/proxy` отправлять журнал изменений в другую БД.

Рассмотрим пример, как можно вести журнал для предыдущего примера в удаленной базе данных `auditdb`:

1. Создайте в удаленной БД таблицу для журнала `emp_audit`.
2. Создайте в удаленной БД функцию `log_emp_audit()`:

```
CREATE FUNCTION log_emp_audit(  
    operation text, userid text, empname text, salary integer  
) RETURNS VOID AS  
$$  
INSERT INTO emp_audit VALUES($1, now(), $2, $3, $4)  
$$ LANGUAGE SQL;
```

3. Создайте прокси-функцию для `log_emp_audit()` в локальной БД (для этого необходимо сначала установить язык `pl/proxy`):

```
CREATE OR REPLACE FUNCTION log_emp_audit(  
    operation text, userid text, empname text, salary integer  
) RETURNS VOID AS $$  
    CONNECT 'dbname=auditdb';  
$$ LANGUAGE plproxy;
```

4. Создайте функции-триггеры, которые будут использовать прокси-функцию для сохранения данных во внешней БД:

```
CREATE OR REPLACE FUNCTION do_emp_audit() RETURNS TRIGGER AS  
$$  
    BEGIN  
        IF (TG_OP = 'DELETE') THEN  
            PERFORM log_emp_audit('DEL', user, OLD.empname, OLD.  
salary);  
        ELSIF (TG_OP = 'UPDATE') THEN  
            -- save old and new values  
            PERFORM log_emp_audit('OLD', user, OLD.empname, OLD.  
salary);  
            PERFORM log_emp_audit('NEW', user, NEW.empname,  
NEW.  
salary);  
        ELSIF (TG_OP = 'INSERT') THEN  
            PERFORM log_emp_audit('INS', user, NEW.empname, NEW.  
salary);  
        END IF;  
        RETURN NULL; -- result is ignored since this is an  
AFTER  
trigger  
    END;  
$$ LANGUAGE plpgsql;
```

5. Добавьте триггеры в таблицу `emp`:

```
CREATE TRIGGER emp_remote_audit  
AFTER INSERT OR UPDATE OR DELETE ON emp  
FOR EACH ROW EXECUTE PROCEDURE do_emp_audit();
```

Убедитесь в том, что база данных аудита защищена. Проследите, чтобы единственным возможным действием пользователя `audit_logger` был вызов функции `log_emp_audit()`.

Подробную информацию о `pl/proxy` вы найдете на странице <http://plproxy.projects.postgresql.org/doc/tutorial.html>.

Интеграция с LDAP

Этот рецепт расскажет вам, как настроить PostgreSQL, чтобы можно было использовать для аутентификации LDAP (Lightweight Directory Access Protocol).

Подготовка

Убедитесь, что имена пользователей вашей БД соответствуют именам на LDAP-сервере, поскольку этот метод работает только для аутентификации пользователей, уже определенных в базе данных.

Как это сделать

В файле аутентификации PostgreSQL `pg_hba.conf` задайте диапазон адресов, для которых будет использоваться метод аутентификации LDAP, а затем сконфигурируйте для них LDAP-сервер:

```
host all 10.10.0.1/16 ldap \
  ldapservers=ldap.our.net ldapprefix='cn=' ldapsuffix=',
dc=our,
dc=net'
```

Как это работает

При таких настройках сервер PostgreSQL будет проверять пароли со сконфигурированного сервера LDAP.

Сервер LDAP не запрашивает права пользователей, однако они должны быть настроены с помощью команд `ALTER USER`, `GRANT` и `REVOKE` в самой базе данных.

И еще

Настройка клиента для использования LDAP

Если вы используете для определения параметров доступа к БД файл `pg_service.conf`, можно сделать так, чтобы некоторые из этих параметров запрашивались LDAP-сервером, добавив в файл `pg_service.conf` следующий код:

```
ldap://ldap.mycompany.com/dc=mycompany,dc=com?uniqueMember
?one?(cn=my
database)
```

А также

О настройке сервера можно прочитать на сайте <http://www.postgresql.org/docs/8.4/static/auth-methods.html#AUTH-LDAP>.

О настройке клиента вы узнаете на странице <http://www.postgresql.org/docs/8.4/static/libpq-ldap.html>.

Соединение с использованием SSL

Здесь рассматривается, как заставить PostgreSQL использовать SSL для защиты соединений путем шифрования всех передаваемых данных. Применение SSL уменьшает вероятность перехвата информации, которая передается между клиентом и БД: имен пользователей, паролей и других секретных данных. Альтернативой использования SSL является соединение по VPN (Virtual Private Network).

Передача данных с применением SSL идет немного медленнее, поэтому в защищенных сетях этот способ лучше не использовать. Особенно неприятным может оказаться влияние на производительность: у вас создается много коротких соединений, поскольку каждая установка SSL-соединения сильно нагружает процессор. В этом случае можно воспользоваться локальным буфером, таким как PgBouncer, с которым вы будете соединяться без шифрования, а затем создавать защищенное SSL-соединение, как описано в FAQ по PgBouncer на сайте <http://pgbouncer.projects.postgresql.org/doc/faq.html>.

Подготовка

Получите или сгенерируйте для сервера SSL-пару ключ-сертификат и сохраните их в директории данных текущей БД в файлы `server.key` и `server.crt`.

Возможно, на некоторых платформах это уже сделали за вас. Например, в Ubuntu PostgreSQL поддерживает SSL-соединения по умолчанию.

Как это сделать

В файле `postgresql.conf` укажите `ssl=on` и перезапустите сервер.

Как это работает

Если указано `ssl=on`, PostgreSQL ждет на один и тот же порт как обычные, так и SSL-соединения (по умолчанию это порт 5432), и определяет тип нового соединения по его первому байту.

После этого при необходимости сервер продолжает устанавливать SSL-соединение.

И еще

Можно предоставить клиенту выбор, какой тип соединения использовать, а можно настроить сервер так, что он будет допускать только SSL-соединения.

В первом случае выполните следующий код:

```
host database user IP-address IP-mask auth-method
```

Чтобы заставить клиентов использовать SSL вместо host, укажите hostssl.

В следующем фрагменте файла pg_hba.conf указано, что сервер будет принимать обычные соединения только из локальной подсети (192.168.1.0/24), но будет требовать SSL для доступа из других сетей.

```
host      all      all      192.168.54.1/32    md5
hostssl   all      all      0.0.0.0/0           md5
```

Получение ключа и сертификата SSL

Для веб-серверов необходимо получать SSL-сертификат в известной организации, выдающей сертификаты (CA, Certificate Authority), так как браузеры, как правило, подозрительно относятся к неизвестному сертификату, и пользователя, который захочет подключиться к серверу с таким сертификатом, ждут «танцы с бубном».

Для своей базы данных лучше сгенерировать сертификат самостоятельно с помощью openssl. Следующие команды генерируют самоподписанный сертификат для вашего сервера:

```
openssl genrsa 1024 > server.key
openssl req -new -x509 -key server.key -out server.crt
```

Подробнее о ключах x509 и сертификатах в openssl рассказывается на сайте <http://www.openssl.org/docs/HOWTO/>.

Настройка клиента для использования SSL

Поведение клиента контролируется переменной окружения PGSSLMODE, которая может иметь следующие значения, определенные в официальных документах PostgreSQL.

MITM означает атаку Man-In-The-Middle, то есть человека, который маскируется под сервер, но на самом деле только подсматривает ваши данные и передает их дальше.

Таблица 6.1.

Режим SSL	Защита от перехвата	Защита от MITM	Описание
disabled	Нет	Нет	Я не думаю о безопасности и не собираюсь тратить ресурсы на шифрование
allow	Возможно	Нет	Я не думаю о безопасности, но буду шифровать информацию, если этого потребует сервер
prefer	Возможно	Нет	Я не думаю о шифровании, но если сервер его поддерживает, то буду шифровать данные
require	Да	Нет	Я хочу шифровать данные, хотя это и повлияет на производительность. Я убежден, что соединяюсь по сети с нужным сервером
verify-ca	Да	Зависит от CA	Я хочу шифровать данные, хотя это влияет на производительность. Я хочу быть уверен, что подключаюсь к доверенному серверу
verify-full	Да	Да	Я хочу шифровать данные и допускаю, что это может повлиять на производительность. Мне важна гарантия, что я подключаюсь к доверенному серверу, который я выбрал

Проверка аутентичности сервера

Два последних режима гарантируют вам, что вы действительно общаетесь со своим сервером, так как проверяется предоставляемый сервером SSL-сертификат.

А также

Больше информации о SSL, в частности об используемой в PostgreSQL библиотеке OpenSSL, можно найти на сайте <http://www.openssl.org> или в книге, посвященной SSL.

Кроме того, на конференции pgcon2009 прозвучал хороший доклад на эту тему, который назывался «Encrypted PostgreSQL» (Зашифрованный PostgreSQL). Слайды этого доклада вы найдете по адресу <http://www.pgcon.org/2009/schedule/events/120.en.html>.

Шифрование секретных данных

В этом рецепте описано шифрование данных с использованием пакета `pgcrypto`.

Подготовка

Узнайте, не запрещено ли шифрование в стране, где вы проживаете или где находится ваш сервер БД (есть и такие страны).

Убедитесь в том, что на узле, где расположена ваша база данных, установлен `pgcrypto`. Например, в Ubuntu он поставляется с пакетом `postgresql-contrib`.

Установите его в БД, в которой будете его использовать.

```
psql mydb < /usr/share/postgresql/8.4/contrib/pgcrypto.sql
```

Кроме того, необходимо будет настроить GPG-ключи:

```
pguser@laptop:~$ gpg --gen-key
```

Здесь ответьте на несколько вопросов, выберите тип ключа DSA and Elgamal и введите пустой пароль.

Теперь экспортируйте ключи:

```
pguser@laptop:~$ gpg -a --export "PostgreSQL User (test key  
for PG Cookbook)" <pguser@somewhere.net>* > public.key  
pguser@laptop:~$ gpg -a --export-secret-keys "PostgreSQL  
User (test  
key for PG Cookbook)" <pguser@somewhere.net>* > secret.key
```

Убедитесь в том, что доступ к секретному ключу имеете только вы и пользователи базы данных `postgres`.

```
pguser@laptop:~$ sudo chgrp postgres secret.key  
pguser@laptop:~$ chmod 440 secret.key  
pguser@laptop:~$ ls -l *.key  
-rw-r--r-- 1 pguser pguser 1718 2010-03-26 13:53 public.key  
-r--r----- 1 pguser postgres 1818 2010-03-26 13:54 secret.  
key
```

Как это сделать

Шифрование

Чтобы секретные ключи не были видны в журналах БД, напишите функцию-обертку, которая будет вытаскивать их из файлов. Это необходимо делать на надежном внутреннем языке, например, `plpythonu`, так как только на доверенном языке можно получить доступ

к файловой системе. Для создания таких функций необходимо быть суперпользователем PostgreSQL.

```
create or replace function get_my_public_key() returns text
as $$
return open('/home/pguser/public.key').read()
$$
language plpythonu;
revoke all on function get_my_public_key() from public;
create or replace function get_my_secret_key() returns text
as $$
return open('/home/pguser/secret.key').read()
$$
language plpythonu;
revoke all on function get_my_secret_key() from public;
```

Если вы не хотите, чтобы пользователи могли видеть фактические ключи, необходимо написать функции для шифрования и дешифровки, а затем дать к ним доступ конечным пользователям.

Функция шифрования работает так:

```
create or replace function encrypt_using_my_public_key(
    cleartext text,
    ciphertext out bytea
)
AS $$
DECLARE
    pubkey_bin bytea;
BEGIN
    -- text version of public key needs to be passed
    through function
    dearmor() to get to raw key
    pubkey_bin := dearmor(get_my_public_key());
    ciphertext := pgp_pub_encrypt(cleartext, pubkey_bin);
END;
$$ language plpgsql security definer;
revoke all on function encrypt_using_my_public_key(text)
from public;
grant execute on function encrypt_using_my_public_key(text)
to bob;
```

Функция дешифровки выполняется так:

```
create or replace function decrypt_using_my_secret_key(
    ciphertext bytea,
    cleartext out text
)
AS $$
DECLARE
    secret_key_bin bytea;
BEGIN
    -- text version of secret key needs to be passed
    through function
```

```
dearmor() to get to raw binary key
    secret_key_bin := dearmor(get_my_secret_key());
    cleartext := pgp_pub_decrypt(ciphertext, secret_key_
bin);
END;
$$ language plpgsql security definer;
revoke all on function decrypt_using_my_secret_key(bytea)
from public;
grant execute on function decrypt_using_my_secret_
key(bytea) to bob;
```

Теперь протестируем шифрование:

```
test2=# select encrypt_using_my_public_key('X marks the
spot!');
```

В результате получаем что-то вроде

```
encrypt_using_my_public_key | \301\301N\003\22
3o\215\2125\203\252;\020\007\376-z\233\211H...
```

Теперь проверим, как это работает в обоих направлениях:

```
test2=# select decrypt_using_my_secret_key(encrypt_using_
my_public_
key('X marks the spot!'));
decrypt_using_my_secret_key
-----
X marks the spot!
(1 row)
```

Все правильно: получили первоначальную строку.

Как это работает

Что вы здесь сделали:

- спрятали ключи от пользователей БД, которые не являются суперпользователями;
- предоставили оболочки для авторизованных пользователей, чтобы использовать шифрование и дешифровку.

Чтобы защитить секретные данные при передаче между клиентом и БД, соединяйтесь с PostgreSQL только по SSL или с локального узла.

Придется довериться администраторам БД и пользователям с правами суперпользователей.

И еще

Для действительно секретных данных

В некоторых случаях лучше не хранить пароль для дешифровки и зашифрованные данные на одной и той же машине.

Целесообразней использовать шифрование с открытым/закрытым ключом и на стороне сервера делать только шифрование. Это означает, что на сервере будет храниться только ключ для шифрования, но не будет ключа для дешифровки. С другой стороны, вы можете встроить в вашу систему серверов отдельный сервер, который будет заниматься только шифрованием/дешифровкой.

Для очень-очень-очень секретных данных

Если информация является еще более секретной и с машины клиента нельзя передавать данные в незашифрованном виде, понадобится шифровать их перед отправкой БД. В таком случае PostgreSQL будет получать информацию сразу в зашифрованном виде, и в обычном он ее не увидит. Это также означает, что вы сможете использовать индексы только для выполнения `WHERE encrypted_column = encrypted_data` и для проверки уникальности. Однако даже `WHERE` = пригодится лишь в том случае, когда алгоритм шифрования каждый раз шифрует один и тот же текст одинаково, что ослабляет защиту.

Две версии pg_crypto

Как правило, `pgcrypto` скомпилирован для использования библиотеки `openssl` (<http://www.openssl.org>). Если по какой-либо причине у вас ее нет или вы не хотите ее использовать, можно скомпилировать версию `pg_crypto` без нее, с небольшим количеством поддерживаемых алгоритмов шифрования и со слегка ухудшенной производительностью.

А также

Страница о `pgcrypto` в документации по PostgreSQL:

<http://www.postgresql.org/docs/9.0/static/pgcrypto.html>.

Сайт об OpenSSL:

<http://www.openssl.org/>.

Документация GNU Privacy Handbook:

<http://www.gnupg.org/gph/en/manual.html>.



ГЛАВА 7.

Администрирование баз данных

Рассматриваемые темы:

- Написание скрипта, который срабатывает полностью или не работает вообще
- Написание скрипта `psql`, который прерывается после первой ошибки
- Действия с несколькими таблицами
- Добавление и удаление столбца
- Изменение типа данных столбца
- Перемещение объектов между схемами
- Добавление и удаление схем
- Добавление и удаление табличных пространств
- Перемещение объектов между табличными пространствами
- Доступ к объектам, находящимся в других БД PostgreSQL
- Обновление представления

Введение

В пятой главе основное внимание было уделено содержимому таблиц и работе с данными. Теперь перейдем на другой уровень и к другим задачам, которые время от времени необходимо выполнять администратору БД: созданию, перемещению, компактному хранению и удалению объектов.

Разумнее всего основные задачи администрирования решать с помощью скрипта, который будет выполнять то, что вам нужно. Сначала нужно запустить его на тестовом сервере, а затем, если все сработает как надо, выполнить его уже на «боевом». Не очень мудро каждый раз вручную вбивать команды на рабочем сервере. Если же вы решите использовать какой-либо инструмент для администрирования, знайте, что он не показывает код, который выполняет. Вы еще случайно не удалили какую-нибудь нужную таблицу? Не сомневайтесь, скоро

это произойдет! Не стоит ли сначала прочитать о резервном копировании? Сделаете резервную копию с помощью скриптов.

Вся прелесть скриптов в том, что вы автоматизируете выполнение общих задач. И не нужно сидеть с мышкой наперевес, продираясь через сотни изменений.

Только не думайте, что это очередной спор на тему, что лучше – командная строка или графический интерфейс. Данная книга предлагает пойти другим путем. Лучший способ избежать множества ошибок и сэкономить время – автоматическое выполнение небольших программ (или скриптов) администрирования. Если бы в скрипте можно было легко и безопасно отразить движения мышкой, то и эта возможность бы рассматривалась. Однако единственный способ написать воспроизводимый скрипт – использовать текстовые команды SQL.

Гораздо интереснее вопрос, чем воспользоваться для создания скриптов. Раз у вас есть PostgreSQL, значит, у вас уже есть `psql`. На нем и остановимся. Все примеры в этой главе написаны для `psql`.

А теперь – к рецептам! Сначала рассмотрим методы скриптования, подходящие для PostgreSQL. Это сделает скрипты более точными и воспроизводимыми и сэкономит время для куда более интересных вещей.

Написание скрипта, который срабатывает полностью или не работает вообще

В задачи администратора БД входит координирование последовательности изменений. И большим плюсом PostgreSQL в данном случае является система транзакций, когда почти все изменения можно выполнить в рамках одной транзакции. Это позволяет создать такой скрипт, который будет выполняться либо полностью, либо никак. Для рабочей системы это может быть очень важно.

Транзакции применяются к DDL (Data Definition Language), то есть к набору команд SQL для определения, изменения и удаления объектов баз данных. Термин DDL уже довольно старый, но его до сих пор используют как удобную замену названий команд, необходимых любому администратору: `CREATE`, `ALTER`, `DROP` и т. п.

Как это сделать

Проще всего убедиться, что выполнены (или не выполнены) все команды скрипта – буквально обернуть скрипт транзакцией:

```
BEGIN;  
command 1;  
command 2;  
command 3;  
COMMIT;
```

Чтобы написать команды контроля транзакции, понадобится отредактировать скрипт, хотя у вас, возможно, нет для этого ни желания, ни возможности. Тогда есть другой способ.

Можно просто в `psql` воспользоваться командами `-l` или `--single-transaction`:

```
bash $ psql -l -f myscript.sql  
bash $ psql --single-transaction -f myscript.sql
```

`-l`, конечно, короче, но лучше использовать `--single-transaction`, поскольку так будет понятнее, что вы делали.

Как это работает

Если хотя бы одна команда в скрипте вызовет сообщение об ошибке или что-то более серьезное, скрипт не будет выполнен. Практически все команды SQL, определяющие объекты (DDL), позволяют избежать ошибок сброса.

Как правило, у команд, начинающихся с `DROP`, есть опция `IF EXISTS`, что позволяет выполнять команду `DROP` вне зависимости от того, существует ли такой объект. После выполнения команды его в любом случае не будет.

```
DROP VIEW IF EXISTS cust_view;
```

Аналогично у команды `CREATE` есть опциональный суффикс `OR REPLACE`. Таким образом, если определение уже существует, его можно переписать, если нет – создать новое:

```
CREATE OR REPLACE VIEW cust_view AS  
SELECT * FROM cust;
```

Можно подумать, что если есть `DROP IF EXISTS` и `CREATE OR REPLACE`, то достаточно использования `CREATE OR REPLACE`. Однако, если вы изменяете определение результата функции или представления, то необходимо сначала использовать `DROP`, а затем `CREATE`:

```
postgres=# CREATE OR REPLACE VIEW cust_view AS  
SELECT col as title1 FROM cust;
```

```
CREATE VIEW
postgres=# CREATE OR REPLACE VIEW cust_view
AS SELECT col as title2 FROM cust;
ERROR: cannot change name of view column "title1" to "title2"
```

Заметим также, что у команды `CREATE INDEX` нет опции `OR REPLACE`. Если выполнить эту команду дважды, у таблицы появятся два индекса (если при этом не указать название индекса). `DROP INDEX IF EXISTS` сработает, но удаление и новое построение индекса может занять слишком много времени.

Обратите внимание на сообщения, которые вызывают следующие команды:

```
postgres=# BEGIN;
BEGIN
postgres=# BEGIN;
WARNING: there is already a transaction in progress
BEGIN
postgres=# COMMIT;
COMMIT
postgres=# COMMIT;
WARNING: there is no transaction in progress
COMMIT
```

PostgreSQL берет первый `BEGIN` (*внешний*) и первый `COMMIT` (*внутренний*). Таким образом, если вы хотите использовать в скрипте команды транзакций, создание обертки более высокого уровня может вызвать проблемы.

Замечание: PostgreSQL допускает, но не выполняет вложенные транзакции. Все команды, которые окажутся после первого `COMMIT`, считаются отдельными транзакциями и будут выполнены, даже если скрипт не сработает. Будьте внимательны!

И еще

В скрипт, использующий транзакции описанным способом, нельзя включить команды:

- `CREATE DATABASE / DROP DATABASE`
- `CREATE TABLESPACE / DROP TABLESPACE`
- `CREATE INDEX CONCURRENTLY`
- `VACUUM`

Но ни одно из этих действий не нужно производить регулярно, вручную или в составе более сложных программ, так что дополнительных проблем возникнуть не должно.

Написание скрипта psql, который прерывается после первой ошибки

По умолчанию скрипт psql после ошибки продолжает выполняться. Звучит странно, но это вызвано только необходимостью исторической совместимости. К счастью, данный факт можно легко обойти.

Подготовка

Начнем с простого скрипта – вы точно знаете, что его команда не выполнится:

```
$ $EDITOR test.sql
mistake1;
mistake2;
mistake3;
```

Выполним этот скрипт в psql:

```
$ psql -f test.sql
psql:test.sql:1: ERROR: syntax error at or near "mistake1"
LINE 1: mistake1;
      ^
psql:test.sql:2: ERROR: syntax error at or near "mistake2"
LINE 1: mistake2;
      ^
psql:test.sql:3: ERROR: syntax error at or near "mistake3"
LINE 1: mistake3;
      ^
```

Как это сделать

Чтобы скрипт прервался после первой ошибки, можно сделать так:

```
$ psql -f test.sql -v ON_ERROR_STOP=on
psql:test.sql:1: ERROR: syntax error at or near "mistake1"
LINE 1: mistake1;
      ^
```

Или добавить в файл test.sql строку:

```
$ $EDITOR test.sql
\set ON_ERROR_STOP
mistake1;
mistake2;
mistake3;
```

Заметим, что это не работает, потому что пропущено ключевое ON:

```
$ psql -f test.sql -v ON_ERROR_STOP
```

Как это работает

`ON_ERROR_STOP` — это специальная переменная, которая определяет, в каком режиме `psql` будет выполнять скрипт. Если она определена, ошибка в SQL будет вызывать в ОС код 3, в то время как остальные ошибки, связанные с ОС, будут вызывать код 1.

И еще

Можно собрать несколько команд `psql` в файл, который будет выполняться при запуске `psql`. Если добавить в этот файл опцию `ON_ERROR_STOP`, она будет применена во всех сессиях `psql`:

```
$ $EDITOR ~/.psqlrc
\set ON_ERROR_STOP
```

Если вам необходимо будет выполнить скрипт без учета созданного файла, воспользуйтесь ключом `-x`. Вероятно, это один из самых безопасных способов при массовом исполнении скриптов.

Действия с несколькими таблицами

Администраторам баз данных часто приходится сталкиваться с ситуацией, когда для решения одной большой задачи необходимо выполнять множество команд. Вам может понадобиться следующее:

- много различных действий со многими таблицами;
- одно и то же действие со многими таблицами;
- одно и то же действие, выполняемое со многими таблицами параллельно;
- различные действия, выполняемые с несколькими таблицами (по одному действию на таблицу) параллельно.

Первый случай является самым общим и требует выполнения скоординированного набора изменений. Как уже говорилось, решением является написание скрипта. Также его можно назвать статическим программированием, так как вы пишете скрипт вручную, а затем исполняете его.

Задачи второго типа решаются с помощью динамических скриптов, то есть вы пишете скрипт, который сам создает скрипт. В данном рецепте рассказывается, как это сделать.

Выполнение действий параллельно неплохо звучит и могло бы быть очень полезным, если бы легко осуществлялось. Иногда это

действительно просто, однако одновременное выполнение нескольких задач с учетом всех ловушек и возможных ошибок гораздо сложнее. Если вы считаете, что за ошибками можно не следить, лучше подумайте еще раз. Запуская задачи параллельно, вы не можете выполнить их в одной транзакции, поэтому проверки на возможные ошибки проводить необходимо. Но не беспокойтесь – как правило, задача параллельного запуска решается намного проще (об этом чуть позже).

Подготовка

Давайте для наших примеров создадим основную схему:

```
postgres=# create schema test;
CREATE SCHEMA
postgres=# create table test.a (col1 INTEGER);
CREATE TABLE
postgres=# create table test.b (col1 INTEGER);
CREATE TABLE
postgres=# create table test.c (col1 INTEGER);
CREATE TABLE
```

Как это сделать

Задача – выполнить такой SQL-код для трех тестовых таблиц, подставив вместо *x* имя таблицы:

```
ALTER TABLE x
ADD COLUMN last_update_timestamp TIMESTAMP WITH TIME ZONE;
```

Для начала напишите скрипт, перечисляющий все таблицы, с которыми вы будете производить какие-либо действия:

```
postgres=# SELECT relname
FROM pg_class c
JOIN pg_namespace n
ON c.renamespace = n.oid
WHERE n.nspname = 'test';
relname
-----
a
b
c
(3 rows)
```

Этот SQL-код сгенерирует текст, подставляющий имя схемы и таблицы в SQL-текст. Полученный результат сохраните в файле `multi.sql`:

```
postgres=# \t on
postgres=# \o multi.sql
```

```
postgres=# SELECT 'ALTER TABLE ' || n.nspname
           || ',' || c.relname ||
           ' ADD COLUMN last_update_timestamp TIMESTAMP WITH TIME
ZONE;'
FROM pg_class c
JOIN pg_namespace n
ON c.relnamespace = n.oid
WHERE n.nspname = 'test';
```

После этого проверьте правильность сгенерированного скрипта:

```
postgres=# \! cat multi.sql
ALTER TABLE test.a ADD COLUMN last_update_timestamp
TIMESTAMP WITH
TIME ZONE;
ALTER TABLE test.b ADD COLUMN last_update_timestamp
TIMESTAMP WITH
TIME ZONE;
ALTER TABLE test.c ADD COLUMN last_update_timestamp
TIMESTAMP WITH
TIME ZONE;
```

а затем запустите его и порауйтесь полученному результату:

```
postgres=# \i multi.sql
ALTER TABLE
ALTER TABLE
ALTER TABLE
```

Как это работает

Это просто пример динамического написания скриптов, которым пользовались администраторы баз данных еще задолго до PostgreSQL.

Он может и не сработать, особенно если сгенерировать SQL-текст с синтаксическими ошибками. Просто исправьте их и будьте внимательны.

Команда `/t` означает «только кортежи», то есть в полученном результате гарантированно не будет заголовков, командных тегов или счетчиков строк.

Команда `/o` отправляет полученный результат в файл.

`!/i` запускает команды операционной системы, поэтому `!/cat` показывает в *nix-системах содержимое файла.

Команда `/i` перенаправляет введенную информацию из файла, то есть выполняет указанный файл. Если скрипт будет выполняться таким образом, вероятно, то, что описывалось в предыдущих рецептах, будут игнорироваться. Так что мы бы рекомендовали придерживаться указаний, данных выше.

Динамическое написание скриптов можно назвать быстрым и «грязным» методом. Такие скрипты не отфильтровывают представления и

другие объекты в тестовой схеме, поэтому при необходимости нужно добавить эту функциональность самостоятельно.

Есть и другой способ:

```
DO $$
DECLARE c record;
BEGIN
    FOR c IN SELECT t.*, n.nspname
    FROM pg_class c JOIN pg_namespace n
    ON c.relnamespace = n.oid
    WHERE n.nspname = 'test' /* ; not needed */
    LOOP
        EXECUTE 'ALTER TABLE ' || quote_ident(n.nspname) ||
            '.' || quote_ident(c.relname) ||
            ' ADD COLUMN last_update_timestamp ' ||
            'TIMESTAMP WITH TIME ZONE; '
    END LOOP;
END $$;
```

Но его лучше не использовать, поскольку в таком случае SQL выполняется непосредственно и это не позволяет сохранить скрипт для дальнейшей работы.

И еще

Как и было обещано, расскажем о параллельном запуске нескольких задач.

Распаралеливание задач можно понимать как разделение основной задачи, когда вы можете запускать x2, x4, x8 и т. п. подписчиков вместо одного большого скрипта.

Для начала следует заметить, что чем больше у вас параллельно запущенных задач, тем труднее отследить ошибки, в то время как прирост производительности заметнее всего для небольшого числа делений. К тому же зачастую вы ограничены мощностью процессора, объемом оперативной памяти и ресурсами ввода/вывода. Это значит, что деление основной задачи на две или четыре параллельных подзадачи неосуществимо без специального инструмента.

Здесь используются два метода – соответственно типу выполняемой задачи:

- задача состоит из небольших подзадач, имеющих приблизительно одинаковый размер;
- задача состоит из небольших подзадач, время выполнения которых различается и зависит от размера/сложности объекта БД.

Если у вас много маленьких задач, вы можете выполнять скрипты много раз, используя циклическое расщепление задач, так что каждый подскрипт будет выполнять половину подзадач. Например:

- Script 1: Add WHERE c. oid % 2 = 0;
- Script 2: Add WHERE c. oid % 2 = 1.

В данном примере требуется добавить в несколько таблиц один столбец. Для столбца не были определены значения по умолчанию, поэтому новый столбец имеет значение NULL, и ALTER TABLE выполняется очень быстро даже для больших таблиц. Если изменить ALTER TABLE, задав значения по умолчанию, необходимо будет переписать всю таблицу. В этом случае время работы будет зависеть от размера таблицы (на самом деле еще и от числа и типа индексов).

Теперь, когда подзадачи имеют разное время выполнения в зависимости от размера, необходимо выделять подзадачи более тщательно, чтобы в итоге получилось несколько скриптов, время работы которых будет примерно одинаковым.

Если известно, что у вас всего несколько больших таблиц, можно просто вручную выделить для каждой отдельный скрипт.

Если в базе данных много больших таблиц, то можно отсортировать SQL-выражения по размеру, а затем, используя циклическое деление, разбить их на несколько подскриптов с примерно одинаковым временем работы. В качестве примера рассмотрим два SQL-выражения:

```
\t on
\o script0.sql
SELECT sql FROM (
SELECT 'ALTER TABLE ' || n.nspname || '.' || c.relname ||
' ADD COLUMN last_update_timestamp TIMESTAMP WITH TIME ZONE
DEFAULT
now();' as sql
,row_number() OVER (ORDER BY pg_relation_size(c.oid))
FROM pg_class c
JOIN pg_namespace n
ON c.relnamespace = n.oid
WHERE n.nspname = 'test'
ORDER BY 2 DESC) as s
WHERE row_number % 2 = 0;
\o script1.sql
SELECT sql FROM (
SELECT 'ALTER TABLE ' || n.nspname || '.' || c.relname ||
' ADD COLUMN last_update_timestamp TIMESTAMP WITH TIME ZONE
DEFAULT
now();' as sql
,row_number() OVER (ORDER BY pg_relation_size(c.oid))
```



```
FROM pg_class c
JOIN pg_namespace n
ON c.relnamespace = n.oid
WHERE n.nspname = 'test'
ORDER BY 2 DESC) as s
WHERE row_number % 2 = 1;
```

Затем выполните две задачи параллельно:

```
$ psql -f script0.sql &
$ psql -f script1.sql &
```

Обратите внимание, как использована функция `row_number()` для сортировки данных по размеру, а затем данные разделены с помощью

```
WHERE row_number % N = i;
```

N — это общее число произведенных скриптов, а i — номер текущего скрипта. Отсчет начинается с нуля, поскольку для распределения подзадач использована modulo arithmetic.

Использование для параллельного запуска задач `pg_batch`

Для параллельного запуска задач существует специальный инструмент, доступный по адресу http://reorg.projects.postgresql.org/pg_batch.html.

Утилита `pg_batch` запускает задачи в порядке нахождения, распределяя их по параллельным сессиям вслепую. Это означает, что необходимо будет написать скрипт, указывающий порядок выполнения задач, чтобы они были распределены между сессиями равномерно. Таким образом, вы все равно придете к тому, что нужно писать скрипты, подобные вышеприведенным.

Добавление и удаление столбца

При работе с таблицами БД администратор часто вынужден добавлять или удалять столбцы. Хотя эти операции являются довольно распространенными, планировать их нужно особенно тщательно, поскольку для их выполнения требуется полная блокировка на довольно длительное время.

Как это сделать

Добавить столбец в таблицу можно следующим образом:

```
ALTER TABLE mytable
ADD COLUMN last_update_timestamp TIMESTAMP WITHOUT TIME ZONE;
```

А удалить – с помощью такой команды:

```
ALTER TABLE mytable  
DROP COLUMN last_update_timestamp;
```

Если нужно выполнить несколько операций, воспользуйтесь `ALTER TABLE`. При этом изменения будут производиться последовательно. Таким образом, вы сможете добавить столбец без ограничений с помощью `IF EXISTS`:

```
ALTER TABLE mytable  
DROP COLUMN IF EXISTS last_update_timestamp,  
ADD COLUMN last_update_timestamp TIMESTAMP WITHOUT TIME ZONE;
```

Заметим, что это почти то же самое, что и

```
UPDATE mytable SET last_update_timestamp = NULL;
```

`ALTER TABLE` работает гораздо быстрее, особенно если требуется обновить данные, но для сохранения уже имеющихся эту команду лучше не использовать.

Как это работает

Для добавления или удаления столбца `ALTER TABLE` полностью блокирует таблицу (на уровне `AccessExclusiveLock`), и никаких других действий с таблицей произвести нельзя. Поэтому нужно выполнить все как можно быстрее.

`DROP COLUMN` фактически не удаляет столбец из всех строк таблицы, а только помечает его как удаленный. Поэтому `DROP COLUMN` – очень быстрая операция.

Операция `ADD COLUMN` также является очень быстрой, если добавлять обнуляемый столбец, имеющий значение по умолчанию `NULL`. Если наложить ограничение `NOT NULL` или задать определенное значение по умолчанию, то нужно будет переписать каждую строку таблицы, и тогда времени понадобится гораздо больше.

Как показано в основном рецепте, `ALTER TABLE` позволяет выполнить сразу много операций со столбцами. Операция `ALTER TABLE` оптимизирована таким образом, что включает все операции со столбцами в один проход по таблице. Это значительно увеличивает скорость при внесении сложной комбинации изменений, например:

```
ALTER TABLE mytable  
ADD COLUMN last_update_userid INTEGER,  
ADD COLUMN last_update_comment TEXT;
```

Если переписать таблицу, то все столбцы, для которых выполнили `DROP`, будут удалены. Если нет, они еще какое-то время будут оставаться

в таблице. Последовательно выполненные операции `INSERT` и `UPDATE` вставят в столбцы после `DROP` значение `NULL`. Таким образом, обновление уменьшит размер хранимых строк, если их значение до сих пор не было нулевым. Теоретически получается, что через некоторое время база данных должна вернуть часть занятого места. Однако на практике это происходит, только если все строки таблицы были обновлены в течение одного периода времени. Во многих таблицах содержатся старые данные, поэтому без дополнительных операций место высвобождено не будет.

В документации по PostgreSQL рекомендуется изменить тип данных столбца на тот же самый, что приведет к переписыванию всех строк. Однако при этом таблица полностью блокируется на длительное время, по крайней мере в больших базах данных. Лучше по возможности не использовать в «боевой» системе для столбцов команду `DROP`. Просто отмечайте те изменения, которые вы бы сделали, если бы было время. В качестве альтернативы можно использовать `VACUUM`, так как при этом таблица не будет переписана (но после `VACUUM FULL` – будет). Однако и здесь следует соблюдать осторожность, поскольку в этом случае таблица также будет блокироваться.

И еще

Индексы, ссылающиеся на удаленный столбец, тоже будут удалены. Если от этих столбцов зависят другие объекты, то `ALTER TABLE` не выполнится. Это можно преодолеть с помощью опции `CASCADE`:

```
ALTER TABLE x
DROP COLUMN last_update_timestamp
CASCADE;
```

Изменение типа данных столбца

К счастью, изменение типов столбцов – операция нечастая. Но если возникла такая необходимость, вы должны учесть все детали, чтобы минимизировать возможность ошибки.

Подготовка

Начнем с простого примера:

```
postgres=# select * from birthday;
name | dob
-----+-----
simon | 690926
(1 row)
```

Эта таблица была создана с помощью

```
CREATE TABLE birthday
( name TEXT
, dob INTEGER);
```

Как это сделать

Допустим, нужно изменить тип данных столбца `dob`. Для начала рассмотрим более простой случай:

```
postgres=# ALTER TABLE birthday
postgres=# ALTER COLUMN dob SET DATA TYPE text;
ALTER TABLE
```

Все сработало. А теперь попробуем снова вернуть тип `integer`, чтобы затем поменять его на более сложный тип `date`:

```
postgres=# ALTER TABLE birthday
postgres=# ALTER COLUMN dob SET DATA TYPE integer;
ERROR: column "dob" cannot be cast to type integer
```

Что пошло не так? Попробуем сконвертировать с помощью `USING`:

```
postgres=# ALTER TABLE birthday
           ALTER COLUMN dob SET DATA TYPE integer
           USING dob::integer;
ALTER TABLE
```

Все работает как надо. Теперь попробуем перейти к типу данных `date`:

```
postgres=# ALTER TABLE birthday
           ALTER COLUMN dob SET DATA TYPE date
           USING date(to_date(dob::text, 'YYMMDD') -
           (case when dob/10000 < 15 then interval '0'
           else interval '100 years' end));
```

Получаем то, что и хотели:

```
postgres=# select * from birthday;
name | dob
-----+-----
Simon | 26/09/1969
(1 row)
```

В PostgreSQL можно также настраивать или сбрасывать выражения по умолчанию вне зависимости от того, наложено ли ограничение `NOT NULL`.

```
ALTER TABLE foo
ALTER COLUMN col DROP DEFAULT expression;
ALTER TABLE foo
ALTER COLUMN col SET DEFAULT 'expression';
ALTER TABLE foo
ALTER COLUMN col SET NOT NULL;
```

```
ALTER TABLE foo
ALTER COLUMN col DROP NOT NULL;
```

Как это работает

Изменение типа с `integer` на `date` потребовало сложного выражения `USING`. Рассмотрим процесс по шагам:

```
postgres=# ALTER TABLE birthday
ALTER COLUMN dob SET DATA TYPE date
USING date(to_date(dob::text, 'YYMMDD') -
(case when dob/10000 <=15 then interval '0'
else interval '100 years' end));
```

Невозможно сразу изменить тип данных с `integer` на `date` — сначала нужно конвертировать данные в `text`, а затем в `date`. `dob::text` означает «преобразование в текст».

После получения `text` инициализирована функция `to_date()`.

Но и этого недостаточно. Исходные данные имели вид 690926 — дата в формате `YYMMDD` (ГГММДД). При конвертировании в формат `date` PostgreSQL предполагает, что двузначное число 69 соответствует году в текущем веке, поэтому на выходе вместо 1969 получается 2069. Добавленное условие `case` означает, что любое значение года, которое больше 15, считается датой прошлого века, то есть из полученного результата вычитается 100.

Любые подобные конвертации следует проверять с помощью `SELECT`. Конвертирование типов данных, особенно в формат дат и обратно, может вызвать затруднения, поэтому не стоит торопиться. Обязательно сделайте резервную копию данных.

И еще

`USING` можно использовать для работы со сложными выражениями, включающими другие столбцы. Это удобно для трансформации данных, например при перебросе в БД с новым строением на рабочем сервере. Соберем все в один пример. Допустим, вам нужно трансформировать следующую таблицу:

```
postgres=# select * from cust;
customerid | firstname | lastname | age
-----+-----+-----+-----
1 | Philip   | Marlowe  | 38
2 | Richard | Hannay  | 42
3 | Molly   | Martins  | 25
4 | Harry   | Palmer   | 36
(4 rows)
```

в таблицу такого вида:

```
postgres=# select * from cust;
customerid | custname      | age
-----+-----+-----
          1 | Philip Marlowe | 38
          2 | Richard Hannay | 42
          3 | Holly Martins  | 25
          4 | Harry Palmer   | 36
(4 rows)
```

Вы можете осуществить это за четыре шага:

```
ALTER TABLE cust ADD COLUMN custname text not null default '';
UPDATE cust SET custname = firstname || ' ' || lastname;
ALTER TABLE cust DROP COLUMN firstname;
ALTER TABLE cust DROP COLUMN lastname;
```

Можно задействовать команды SQL непосредственно или прибегнуть к помощи инструмента вроде pgAdmin3. Однако выполнение приведенного кода, скорее всего, вызовет затруднения: ведь если изменения не совершаются в одной транзакции, другие пользователи могут видеть их, даже если они выполнены только наполовину. Так что лучше вносить эти изменения в одной транзакции, заключив их в `BEGIN` и `COMMIT`. Кроме того, эти четыре изменения потребуют двух проходов по таблице. Однако можно обойтись и одним проходом, так что лучше сделать следующее:

```
BEGIN;
ALTER TABLE cust
  ALTER COLUMN firstname SET DATA TYPE text
  USING firstname || ' ' || lastname,
  ALTER COLUMN firstname SET NOT NULL,
  DROP COLUMN lastname;
ALTER TABLE cust RENAME firstname TO custname;
COMMIT;
```

Это хороший пример того, почему предпочтительней использовать в больших базах данных скрипты, а не вносить изменения с помощью GUI.

Добавление и удаление схем

Хороший способ улучшить эффективность администрирования – разделение объектов. Осталось только узнать, как создавать новые схемы и удалять те, которые больше не нужны.

Как это сделать

Для добавления новой схемы выполните команду:

```
CREATE SCHEMA sharedschema;
```

Если нужно, чтобы этой схемой владел определенный пользователь, можно добавить соответствующую опцию:

```
CREATE SCHEMA sharedschema AUTHORIZATION scarlett;
```

Если нужно создать схему, имя которой совпадает с именем существующего пользователя, чтобы этот пользователь стал ее владельцем, попробуйте выполнить следующее:

```
CREATE SCHEMA AUTHORIZATION scarlett;
```

Во многих СУБД имя схемы совпадает с именем ее владельца. В PostgreSQL в схеме, принадлежащей одному пользователю, могут быть объекты, принадлежащие другому пользователю, поэтому если имя схемы совпадает с именем ее владельца, возникнет путаница. Ее можно избежать, если использовать два типа схем: в схеме, имя которой совпадает с именем ее владельца, должны находиться только объекты, принадлежащие тому же пользователю; у других схем допустимо разделяемое владение. Других дополнительных опций у команды `CREATE SCHEMA` нет.

Для удаления схемы с именем `str` нужно выполнить:

```
DROP SCHEMA str;
```

Заметим, что команды `CREATE OR REPLACE SCHEMA` нет, поэтому если вы хотите создать схему (вне зависимости от того, существует ли она), потребуется выполнить следующее:

```
DROP SCHEMA IF EXISTS newschema;  
CREATE SCHEMA newschema;
```

`DROP SCHEMA` работает только для пустой схемы или если использовать эту команду вместе с ядерной опцией:

```
DROP SCHEMA IF EXISTS newschema CASCADE;
```

Ядерный вариант убивает все известные микробы и все объекты вашей БД.

И еще

В стандарте SQL допускается создать схему и объекты, ее населяющие, одним SQL-выражением. При желании вы можете воспользоваться таким синтаксисом и в PostgreSQL:

```
CREATE SCHEMA foo  
CREATE TABLE account  
(id INTEGER NOT NULL PRIMARY KEY  
,balance NUMERIC(50,2))  
CREATE VIEW accountsample AS  
SELECT *
```

```
FROM account  
WHERE random() < 0.1;
```

Впрочем, по мнению многих, при использовании этого синтаксиса возникают дополнительные ограничения. С его помощью можно создавать сразу несколько объектов, а в PostgreSQL проще воспользоваться транзакционными DDL, о которых уже говорилось ранее.

Права на уровне схем

С помощью команды `GRANT` можно назначить права на объекты схемы:

```
GRANT SELECT ON ALL TABLES IN SCHEMA sharedschema TO PUBLIC;
```

Однако она повлияет только на уже созданные таблицы, а таблицы, которые будут формироваться позже, унаследуют права, определенные командой `ALTER DEFAULT PRIVILEGES`, например:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA sharedschema  
GRANT SELECT ON TABLES TO PUBLIC;
```

Перемещение объектов между схемами

Схемы созданы, и теперь вы, скорее всего, захотите переместить существующие объекты для поддержания порядка.

Как это сделать

Переместить таблицу из одной схемы в другую можно с помощью:

```
ALTER TABLE cust  
SET SCHEMA otherschema;
```

Если вы хотите переместить все объекты, вероятно, достаточно будет просто переименовать схему:

```
ALTER SCHEMA existingschema RENAME TO otherschema;
```

Переименование сработает только в том случае, если схемы с таким именем еще не было.

Иначе для каждой таблицы, которую вы хотите перенести, необходимо будет выполнить `ALTER TABLE`. Здесь можно также воспользоваться рецептом, рассказывающим, как выполнять действия с несколькими таблицами.

Представления, последовательности, функции, агрегатные функции, домены также можно переместить командой `ALTER` с опцией `SET SCHEMA`.

Как это работает

При перемещении таблицы в новую схему все индексы, триггеры и правила, определенные на этих таблицах, также будут перемещены в новую схему. Если вы использовали тип данных `SERIAL` и создали неявную последовательность, то она тоже будет перемещена в новую схему. Применение схем важно только для администрирования и никак не повлияет на расположение файлов данных таблиц. Из последующих рецептов вы узнаете, что табличные пространства работают совершенно по-другому.

Базы данных, пользователи/роли, языки, конвертации в схемах не представлены. Схема существует в определенной БД. Схемы не могут быть расположены в других схемах: они не встроены в дерево или иерархию.

И еще

В определенной схеме могут существовать объекты текстового поиска, однако нет команд для перемещения их в другие схемы. Аналогично в схеме могут быть операторы, классы операторов или семейства операторов, однако команд для перемещения их в новую схему нет. Кроме того, касты не принадлежат схеме – в отличие от типов данных и функций, на которые они ссылаются (подробное рассмотрение выходит за рамки данной книги, поэтому здесь приводится просто справка для тех, кому интересно, как все работает).

Добавление и удаление табличных пространств

Табличные пространства позволяют хранить данные PostgreSQL на разных устройствах. Такое разделение используется для повышения производительности и упрощения администрирования. В конце концов, база данных может просто не поместиться на одном диске.

Подготовка

Прежде чем приступить к созданию полезного табличного пространства, необходимо подготовить к работе соответствующие устройства.

Следует задуматься о скорости, объеме и надежности дисков, которые вы собираетесь использовать. Убедитесь в том, что они

правильно сконфигурированы. Эти решения могут сильно повлиять на вашу жизнь в последующие несколько месяцев или лет.

О производительности дисков многие даже не задумываются. Настоятельно рекомендуем прочитать десятую главу этой книги, а также ознакомиться с другими книгами по данной теме.

После предварительной подготовки можно создавать директорию для табличного пространства. Эта директория должна:

- быть пустой;
- принадлежать пользователю, которому принадлежит сервер PostgreSQL;
- определяться с помощью абсолютного пути.

В системах Linux/Unix не следует напрямую использовать точку монтирования. Вместо этого создайте поддиректорию и используйте ее. Это упрощает владение и позволяет избежать некоторых проблем на уровне файловой системы.

Кроме того, следует подбирать имена так, чтобы было понятно, какое табличное пространство с каким сервером связано. Не ограничивайтесь чем-то слишком простым, например `data`, поскольку в дальнейшем это может затруднить администрирование. Кроме того, следует помнить, что серверы для тестов или для разработки не могут и не должны смешиваться с «боевыми» системами.

Как это сделать

После создания директории создать табличное пространство несложно:

```
CREATE TABLESPACE new_tablespace  
LOCATION '/usr/local/pgsql/new_tablespace';
```

Команда для удаления табличного пространства также очевидна:

```
DROP TABLESPACE new_tablespace;
```

Можно удалить только пустое табличное пространство, но как узнать, является ли оно пустым?

В табличных пространствах могут быть постоянные и временные объекты. Постоянные объекты данных – это таблицы, индексы и объекты `toast` (The Oversized Attribute Storage Technic – метод хранения больших значений). Но об объектах `toast` можно не беспокоиться, так как они создаются и всегда «живут» в том же табличном пространстве, что и их основная таблица; к тому же, правами на них вы управлять все равно не сможете.

Индексы могут находиться в разделенных табличных пространствах, если это нужно для повышения производительности. Для

этого требуется отдельное указание при создании индекса с помощью `CREATE INDEX`. По умолчанию индексы создаются в том же табличном пространстве, что и таблица, которой они принадлежат.

В табличном пространстве могут быть и временные объекты – специально созданные пользователями временные таблицы или файлы данных, создаваемые неявно, когда большие запросы не помещаются в указанное для них пространство `work_mem`. Они создаются с учетом значения параметра `temp_tablespace`, и это, как правило, вызывает определенные трудности. Дело в том, что пользователи могут изменять для себя значение данного параметра, и вы никогда не узнаете, какое значение он имеет для каждого пользователя.

Табличное пространство для каждого объекта пользователя можно выяснить с помощью запроса:

```
SELECT          spcname
                ,relname
                ,CASE WHEN relistemp THEN 'temp '
CASE
                WHEN relkind = 'r' THEN 'table'
                WHEN relkind = 'v' THEN 'view'
                WHEN relkind = 'S' THEN
'sequence'
                WHEN relkind = 'c' THEN 'type'
                ELSE 'index' END as objtype
FROM            pg_class c join pg_tablespace ts
ON              (CASE WHEN c.reltablespace = 0 THEN
                (SELECT dattablespace FROM pg_
database
                WHERE datname = current_
database())
                ELSE c.reltablespace END) = ts.oid
WHERE           relname NOT LIKE 'pg_toast%'
AND            relnamespace NOT IN (SELECT oid FROM pg_
namespace WHERE
nspname IN ('pg_catalog', 'information_schema'))
;
spcname          | relname   | objtype
-----+-----+-----
new_tablespace   | x         | table
new_tablespace   | y         | table
new_tablespace   | z         | temp table
new_tablespace   | y_val_idx | index
```

Можно также посмотреть значения в столбцах `spcowner`, `relowner`, `relacl` и `spcacl`, чтобы узнать, кто владелец и каковы его права. Столбцы `relacl` и `spcacl` ссылаются на «список контроля доступа», в котором определены права доступа к этим объектам. В столбцах `spcowner`

и `relowner` записан владелец табличного пространства и таблиц/индексов соответственно.

Как это работает

Табличное пространство – это директория, где хранятся файлы данных PostgreSQL. Вы используете символичные ссылки из директории данных в табличное пространство.

Toast-таблицы исключаются, поскольку они всегда находятся в том же табличном пространстве, что и исходные таблицы, хотя toast-таблицы всегда принадлежат другой схеме. Эти таблицы можно исключить с помощью столбца `relkind`, однако при этом все еще будут задействованы индексы toast-таблиц. Однако toast-таблицы и toast-индексы легко исключить из запроса, поскольку они всегда начинаются с `pg_toast`.

Предыдущий запрос оказался таким сложным, поскольку `pg_class` для объекта показывает `reltablespace = 0`, если объект был создан в табличном пространстве, которое для этой базы данных является значением по умолчанию. Поэтому, если напрямую соединить `pg_class` и `pg_tablespace`, будут потеряны строки.

Обратите внимание, что вы можете видеть временные объекты и узнать, в каком табличном пространстве они были созданы, хотя у вас не получится сослаться на них из сессии другого пользователя.

И еще

В табличном пространстве могут находиться объекты из нескольких баз данных, поэтому из текущей БД их не видно. Однако табличное пространство не удаляется, и вы получаете сообщение об ошибке:

```
ERROR: tablespace "old_tablespace" is not empty
```

Чтобы избежать этой проблемы, используйте для каждой БД отдельное табличное пространство. Это особенно важно, если у вас в различных базах данных есть таблицы и схемы с одинаковыми именами. Если вы создали новую директорию для табличного пространства, в ней можно создать поддиректории для каждой таблицы и использовать эти поддиректории в табличных пространствах.

При необходимости присвойте каждому табличному пространству определенного владельца, выполнив запрос:

```
ALTER TABLESPACE new_tablespace OWNER TO eliza;
```

Кроме того, можно настроить для пользователя `default_tablespace`, так чтобы его таблицы по умолчанию создавались именно там:

```
ALTER USER eliza SET default_tablespace = 'new_tablespace';
```

Выделение pg_xlog на отдельное устройство

Чтобы улучшить производительность, вам, возможно, потребуется поместить директорию `pg_xlog` на отдельное устройство. Казалось бы, похоже на табличные пространства, однако специальной команды для такого перемещения при работающем сервере нет. Рецепт вы найдете в десятой главе.

Настройка на уровне табличных пространств

У каждого табличного пространства свои характеристики ввода/вывода, поэтому вы можете изменить для каждого табличного пространства параметры стоимости планирования. Например:

```
ALTER TABLESPACE new_tablespace SET  
(seq_page_cost = 0.05, random_page_cost = 0.1);
```

Эти настройки соответствуют SSD-диску, то есть скорость случайного считывания в 10 раз, а последовательного считывания – в 20 превосходит аналогичные параметры жесткого диска.

Данные значения можно было бы обсудить подробнее, но, к сожалению, это выходит за рамки данной книги.

Перемещение объектов между табличными пространствами

Иногда требуется перенести данные из одного табличного пространства в другое.

Подготовка

Сначала создайте эти табличные пространства. Если у вас есть старое и новое табличное пространство, можно выполнять команды для перемещения.

Как это сделать

В табличном пространстве могут находиться постоянные и временные объекты. Постоянные объекты данных – это таблицы, индексы

и toast-объекты. Но toast-объекты можно отдельно не рассматривать. Они всегда создаются и находятся в том же табличном пространстве, что и их основная таблица, поэтому если ее переместить в новое табличное пространство, toast-объекты будут перемещены вместе с ней.

```
ALTER TABLE mytable SET TABLESPACE new_tablespace;
```

Индексы могут находиться в отдельных табличных пространствах, и при перемещении таблицы индексы останутся там, где были. Не забудьте для каждого индекса выполнить команду `ALTER INDEX`:

```
ALTER INDEX mytable_val_idx SET TABLESPACE new_tablespace;
```

Временные объекты переместить нельзя – говоря о перемещении, имеют в виду, что в дальнейшем они будут создаваться в новом месте. Для этого необходимо:

1. Изменить `temp_tablespace`.
2. Перегрузить сервер, чтобы новые настройки вступили в силу.

Одной команды, работающей для всех пользователей, нет.

Как это работает

Если вы хотите переместить таблицу и ее индексы одним действием, можно выполнить все команды в одной транзакции:

```
BEGIN;  
ALTER TABLE mytable SET TABLESPACE new_tablespace;  
ALTER INDEX mytable_val1_idx SET TABLESPACE new_tablespace;  
ALTER INDEX mytable_val2_idx SET TABLESPACE new_tablespace;  
COMMIT;
```

Перемещение табличных пространств означает простое копирование больших объемов данных. Копирование производится постепенно, по блокам, и, к сожалению, чем больше таблица, тем больше времени оно займет. Чтобы повысить производительность, отключите архивирование и потоковую репликацию, поскольку тогда не будут вестись записи в WAL.

Следует также помнить, что на время копирования таблица полностью блокируется (`AccessExclusiveLock`), а это может привести к остановке работы приложения. Будьте внимательны!

Чтобы объект при следующем его создании появился в нужном месте, воспользуйтесь командой:

```
SET default_tablespace = 'new_tablespace';
```

Не следует путать с командой

```
ALTER DATABASE mydb SET TABLESPACE new_tablespace;
```

поскольку в этом случае все объекты, для которых не было определено табличное пространство, будут перемещены в `new_tablespace`. Если база данных большая, времени на это понадобится много, и БД будет заблокирована надолго. Особенно неприятно, если это вышло случайно.

И еще

Если вы вдруг обнаружили, что индексы не перемещаются при перемещении таблицы, проверьте, находятся ли они в том же табличном пространстве, что и их таблицы. Для этого выполните:

```
SELECT          i.relname as indexname
                ,tsi.spcname
                ,t.relname as tablename
                ,tst.spcname
FROM            (((pg_class t /* tables */
JOIN pg_tablespace tst
ON t.reltablespace = tst.oid)
JOIN pg_index pgi
ON pgi.indrelid = t.oid)
JOIN pg_class i /* indexes */
ON pgi.indexrelid = i.oid)
JOIN pg_tablespace tsi
ON i.reltablespace = tsi.oid
WHERE           i.relname NOT LIKE 'pg_toast%'
AND             i.reltablespace != t.reltablespace
;
```

Если таблица и индекс находятся в разных табличных пространствах, это можно увидеть в `psql`:

```
postgres=# \d y
Table "public.y"
Column | Type | Modifiers
-----+-----+-----
val     | text |
Indexes:
    "y_val_idx" btree (val), tablespace "new_tablespace"
Tablespace: "new_tablespace2"
```

В результате представленного запроса вы получите:

```
relname      | spcname      | relname | spcname
-----+-----+-----+-----
y_val_idx    | new_tablespace | y       | new_tablespace2
(1 row)
```

Доступ к объектам, находящимся в других БД PostgreSQL

Иногда требуется получить доступ к данным, находящимся в других БД PostgreSQL. Причины могут быть разными:

- у вас несколько серверов, и необходимо взять какие-либо данные с одного сервера (например, справочные) и подгрузить их на другом;
- вам необходимы данные из другой БД на том же сервере, которые перенесены туда по административным причинам;
- вам нужно выполнить действия, которые не должны быть отменены в случае возникновения ошибки или прерывания транзакции. Их еще называют *побочными функциями* или *автономными транзакциями*.

Если вы заинтересовались этим вопросом при изучении масштабирования, шардинга (горизонтального разбиения) или методов балансирования нагрузки, перейдите сразу к последнему разделу этого рецепта, а затем к двенадцатой главе.

Подготовка

Установите контриб-модуль `dblink`.

Затем создайте несколько определений доступа. Для этого есть несколько способов, но сейчас воспользуемся командами SQL Standard (SQL/MED):

```
postgres=# CREATE FOREIGN DATA WRAPPER postgresql
VALIDATOR postgresql_fdw_validator;
CREATE FOREIGN DATA WRAPPER
postgres=# CREATE SERVER otherdb
FOREIGN DATA WRAPPER postgresql
OPTIONS (host 'foo', dbname 'otherdb', port '5432');
CREATE SERVER
postgres=# CREATE USER MAPPING FOR PUBLIC
SERVER otherdb;
CREATE USER MAPPING
```

Достаточно одной FOREIGN DATA WRAPPER, однако SERVER необходимо создать для каждой базы данных PostgreSQL, с которой вы будете устанавливать соединение. Это только определение соединения, но не само соединение.

Создание `public user mapping` без дополнительных условий кажется странным, но это будет означать, что вы используете по умолчанию модель поведения, соответствующую `libpq`. То есть вы будете устанавливать соединение с удаленной БД под именем `PGUSER` или, если такого пользователя нет, – как пользователь операционной системы.

Как это сделать

Создать неназванное соединение можно следующим образом:

```
SELECT dblink_connect('otherdb');
dblink_connect
-----
OK
(1 row)
```

Чтобы разорвать неназванное соединение, выполните:

```
SELECT dblink_disconnect();
dblink_connect
-----
OK
(1 row)
```

Чтобы выполнить следующую команду:

```
postgres=# INSERT INTO audit_log VALUES (current_user,
now());

INSERT 0 1
```

запустите ее по неназванному удаленному соединению:

```
postgres=# SELECT dblink_exec('INSERT INTO audit_log
VALUES'||
' (current_user, now())', true);
dblink_exec
-----
INSERT 0 1
(1 row)
```

Обратите внимание: удаленная команда возвращает обозначение команды и число обработанных строк. Второе условие означает «выход при ошибке». Если присмотреться, видна еще одна возможная ошибка: когда `INSERT` выполняется локально, на сервере берется текущее значение `current_user`, однако при удаленном выполнении вы используете значение `current_user` с удаленного сервера, которое может отличаться в зависимости от определенного перед этим значения `user mapping`.

Следующий запрос

```
SELECT generate_series(1,3)
```

по неназванному удаленному соединению выполняется так:

```
SELECT *
FROM dblink('SELECT generate_series(1,3)'
  AS link(col1 integer);
 col1
-----
    1
    2
    3
(3 rows)
```

Заметим, что необходимо указать результирующие столбцы и их типы. Если `dblink()` не может определить спецификацию результата запроса, она будет выполнена после парсинга запроса, поэтому обязательно нужно явно обозначать, что вы ожидаете увидеть на выходе после выполнения запроса.

Как это работает

`dblink` устанавливает постоянное соединение с другой базой данных. Функции `dblink` отслеживают параметры этого соединения, так что об этом можно не беспокоиться. Помните о том, что вы работаете с внешним ресурсом, чтобы не возникла типичная ошибка «утечки ресурсов»: если забыть о том, что установлено удаленное соединение и не разорвать его вовремя, в дальнейшем могут возникнуть проблемы. Удаленные соединения будут разорваны при прерывании вашей сессии.

Удаленное соединение поддерживается даже в случае прерывания транзакций и других ошибок, поэтому нет необходимости переподключаться.

`dblink()` выполняет удаленный запрос и держит итоговую выборку в памяти, а лишь затем отправляет локальный ответ. Таким образом, большие запросы могут не выполняться из-за недостатка памяти. Однако указанный метод не предназначен для работы с большими потоками данных. Если это именно ваш случай, прочитайте рецепт подгрузки данных.

Более объемные запросы можно осуществить с помощью курсоров. Это позволяет получать ответ маленькими порциями. Следует открыть курсор, затем в цикле отобрать подходящие строки, пока не будут найдены все необходимые, а затем закрыть курсор. Например:

[illegible]

```

dblink_open
-----
OK
(1 row)
postgres=# SELECT *
                                FROM dblink_fetch('example', 10,
true)
                                AS link (coll integer);

 coll
-----
    1
    2
    3
(3 rows)

```

Обратите внимание, что при открытии курсора нет необходимости его определять, однако следует задать результаты, которые нужно будет выбрать, так же, как и с обычными запросами. Вызываем по 10 строк.

```

postgres=# SELECT *
                                FROM dblink_fetch('example', 10, true)
                                AS link (coll integer);

 coll
-----
(0 rows)
postgres=# SELECT dblink_close('example');
dblink_close
-----
OK
(1 row)

```

`dblink` также позволяет использовать сразу несколько соединений. Работа только с одним соединением не является оптимальным для модульного программирования. В более сложных ситуациях можно предполагать, что нужное вам соединение – это не то, которое может понадобиться другой части программы. `dblink` разрешает давать соединениям имена, так что вам не нужно будет угадывать, является ли соединение по умолчанию именно тем, которое вам нужно. К тому же есть функция `dblink_get_connections()`, с помощью которой можно посмотреть список активных соединений.

И еще

Кажется, что удаленные источники данных можно считать таблицами, однако в реальности это не совсем то, чего можно было бы ожидать.

Нет никакого интегрированного оптимизатора запросов. Если присоединять (`join`) удаленную таблицу к локальной, данные из удаленной БД будут подтягиваться напрямую, даже в том случае, когда

быстрее было бы сначала подгрузить данные, а затем вытянуть из них нужные строки.

Локальное предложение `WHERE` удаленной базе не отправляется, поэтому подобные запросы будут выполняться неправильно:

```
SELECT *
FROM dblink('otherdb',
            'SELECT * FROM bigtable') AS link ( ... )
WHERE filtercolumn > 100;
```

Нужно непосредственно добавлять `WHERE` в удаленный запрос:

```
SELECT *
FROM dblink('otherdb',
            'SELECT * FROM bigtable' ||
            ' WHERE filtercolumn > 100') AS link ( ... );
```

и это приводит к мысли, что настройка представлений для удаленных данных не так уж нужна. Она заставляет пользователей думать, что расположение таблицы не имеет значения, хотя с точки зрения производительности это определенно не так. То же самое можно сказать и о других интегрированных продуктах или продуктах для удаленного доступа к базам данных.

Кроме того, возникает еще несколько вопросов, связанных с производительностью. Во-первых, при выполнении удаленных запросов текущая сессия ожидает их завершения. Можно также выполнять запросы, не ожидая, пока они завершатся и вернут какой-либо ответ, с помощью функций:

- `dblink_send_query();`
- `dblink_is_busy();`
- `dblink_get_result();`

Если вы переживаете по поводу превышения времени соединения, можно использовать пул сессий. Таким образом, некоторое число соединений будет зарезервировано, что позволит вам сократить время соединения. Рецепты, относящиеся к пулам соединений, вы найдете в четвертой главе.

А также

Еще один способ доступа к другим базам данных представляет инструмент `PL/Proxy`. `PL/Proxy` позволяет создавать локальную функцию, являющуюся прокси-функцией для удаленной БД. `PL/Proxy` работает только с функциями, что многие воспринимают как серьезное ограничение. Именно поэтому основная часть этого раздела была посвящена `dblink`.

Создать локальную прокси-функцию несложно:

```
CREATE FUNCTION my_task(VOID)
RETURNS SETOF text AS $$
    CONNECT 'dbname=myremoteserver';
    SELECT my_task();
$$ LANGUAGE plproxy;
```

Вам требуется локальная функция, но не нужно вызывать удаленную функцию, поэтому можно использовать SQL-высказывания непосредственно. Пример параметризованной функции:

```
CREATE FUNCTION get_cust_email(p_username text)
RETURNS SETOF text AS $$
    CONNECT 'dbname=myremoteserver';
    SELECT email FROM users WHERE username = p_username;
$$ LANGUAGE plproxy;
```

PL/Proxy была разработана специально для создания более сложной архитектуры для шардинга и балансировки нагрузки. Команда `RUN ON` позволяет динамически задавать удаленную базу данных, для которой вы будете выполнять SQL-выражение. Таким образом, предыдущий пример будет выглядеть так:

```
CREATE FUNCTION get_cust_email(p_username text)
RETURNS SETOF text AS $$
    CLUSTER 'mycluster';
    RUN ON hashtext(p_username);
    SELECT email FROM users WHERE username = p_username;
$$ LANGUAGE plproxy;
```

Прежде чем вы начнете создавать архитектуру, пользуясь этими понятиями, прочитайте двенадцатую главу.

Обновление представления

PostgreSQL поддерживает команду SQL-стандарта `CREATE VIEW`, однако создаваемые таким образом представления не являются обновляемыми. Возможно, в дальнейших версиях это будет изменено, но в 9.0 администратор все еще может столкнуться с этой проблемой.

Подготовка

Сначала необходимо понять, что производить операции вставки, удаления или обновления легко только с простыми представлениями. Стандарт SQL различает представления, которые являются легко обновляемыми, и более сложные представления.

Итак, рассмотрим, какое представление можно считать легко обновляемым, а какое нет. Начнем с таблицы `cust`:

```
postgres=# SELECT * FROM cust;
customerid | firstname | lastname | age
-----+-----+-----+-----
          1 | Philip    | Marlowe   | 38
          2 | Richard  | Hannay    | 42
          3 | Holly    | Martins   | 25
          4 | Harry    | Palmer    | 36
          4 | Mark     | Hall      | 47
(5 rows)
```

Создадим на ее основе очень простое представление:

```
CREATE VIEW cust_view AS
SELECT customerid
       ,firstname
       ,lastname
       ,age
FROM cust;
```

Каждая строка этого представления соответствует строке таблицы-источника, ссылка на каждый столбец дается непосредственно, без вызова функции. Таким образом, вы сможете через созданное представление применять операции `insert`, `update` и `delete` к исходной таблице.

Ниже представлен пример представления, операции с которым так легко перенести на исходную таблицу:

```
CREATE VIEW cust_avg AS
SELECT avg(age)
FROM cust;
CREATE VIEW cust_above_avg_age AS
SELECT customerid
       ,substr(firstname, 1, 20) as fname
       ,substr(lastname, 1, 20) as lname
       ,age -
       (SELECT avg(age)::integer
        FROM cust) as years_above_avg
FROM cust
WHERE age >
      (SELECT avg(age)
       FROM cust);
CREATE VIEW potential_spammers AS
SELECT customerid
FROM cust
ORDER BY spam_score(firstname, lastname) DESC
LIMIT 100;
```

Итак, прежде чем заняться выделением прав для всех или отдельных пользователей на внесение изменения в таблицу через представления, определитесь, действительно ли в этом есть необходимость.

Как это сделать

В PostgreSQL существует способ создания правил переписывания запросов. Что-то подобное есть в Oracle – instead-of triggers, однако в других СУБД аналогов этой функциональности нет.

Начнем с простого представления, которое может понадобиться для администрирования:

```
CREATE VIEW cust_view AS
SELECT customerid
       ,firstname
       ,lastname
       ,age
FROM cust;
```

Если попробовать применить INSERT, будет выдано сообщение об ошибке:

```
postgres=# INSERT INTO cust_view
postgres=# VALUES (5, 'simon', 'riggs', 133);
ERROR:  cannot insert into a view
HINT:  You need an unconditional ON INSERT DO INSTEAD rule.
```

Для начала создайте правило:

```
CREATE RULE cust_view_insert AS
ON insert TO cust_view
DO INSTEAD
INSERT INTO cust
VALUES (new.customerid, new.firstname, new.lastname, new.age);
```

А теперь еще раз попробуйте сделать INSERT:

```
postgres=# INSERT INTO cust_view
postgres=# VALUES (5, 'simon', 'riggs', 133);
INSERT 0 1
```

Теперь все работало. Добавьте также правила для операций DELETE и UPDATE:

```
CREATE RULE cust_view_update AS
ON update TO cust_view
DO INSTEAD
UPDATE cust SET
  firstname = new.firstname
,lastname = new.lastname
,age = new.age
WHERE customerid = old.customerid;
CREATE RULE cust_view_delete AS
ON delete TO cust_view
DO INSTEAD
DELETE FROM cust
WHERE customerid = old.customerid;
```

Как это работает

Это только крошечная часть того, чего можно достичь, используя такие правила, однако они слишком сложны для широкого применения.

Хотя с помощью правил можно сделать очень многое, необходимо понимать, имеет ли это смысл и практическую ценность в данном конкретном случае. Есть и еще кое-что, о чем следует упомянуть, прежде чем вы увлечетесь и начнете использовать правила везде, где только возможно.

PostgreSQL применяет правила после того, как SQL-код был получен и проверен на наличие ошибок, но до того, как планировщик попытается оптимизировать SQL-выражение.

В правилах этого рецепта для строк использовались значения `old` и `new`, как и с триггерными функциями. При этом в случае с `INSERT` применялись только новые значения, а в случае с `DELETE` — только старые.

Большой недостаток правил заключается в невозможности прямой подгрузки данных в таблицу с помощью команды `COPY`. Невозможно трансформировать последовательность `INSERT` в одну команду `COPY` и нельзя применить `COPY` к представлению. Для прямой подгрузки данных требуется непосредственный доступ к таблице.

Если используется такое представление:

```
CREATE VIEW cust_minor AS
SELECT customerid
       ,firstname
       ,lastname
       ,age
FROM cust
WHERE age < 18;
```

то возникает дополнительная сложность. Если потребуется его обновить, то вы, прочитав документацию, увидите, что необходимо добавить правило, отражающее условие `WHERE`:

```
CREATE RULE cust_minor_update AS
ON update TO cust_minor
WHERE new.age < 18
DO INSTEAD
UPDATE cust SET
    firstname = new.firstname
    ,lastname = new.lastname
    ,age = new.age
WHERE customerid = old.customerid;
```

Однако это не сработает. Нужно добавить два правила, одно из которых не является условием и буквально ничего не делает, будучи

необходимым только по внешним причинам, а другое будет выполнять всю необходимую работу:

```
CREATE RULE cust_minor_update_dummy AS
ON update TO cust_minor
DO INSTEAD NOTHING;
CREATE RULE cust_minor_update_conditional AS
ON update TO cust_minor
WHERE new.age < 18
DO INSTEAD
UPDATE cust SET
    firstname = new.firstname
    ,lastname = new.lastname
    ,age = new.age
WHERE customerid = old.customerid;
```

И еще

Некоторых администраторов БД эти правила сильно раздражают. Почему? Попробуйте решить задачу из рассмотренного примера другим способом – с помощью триггеров. Сделайте представление обновляемым:

```
CREATE VIEW cust_view AS
SELECT customerid
    ,firstname
    ,lastname
    ,age
FROM cust;
```

Нельзя создать триггер для представления, поэтому вместо этого создайте таблицу:

```
CREATE TABLE cust_view AS SELECT * FROM cust WHERE 1 = 0;
```

Эмулируйте представление, создав правило `SELECT` для этой таблицы, а затем попробуйте написать для этой таблицы триггеры для действий `INSERT`, `UPDATE` и `DELETE`. Правило выполнится только для пустой таблицы, и то если оно называется `_RETURN`.

```
postgres # CREATE RULE "_RETURN" AS
          ON SELECT TO cust_view
          DO INSTEAD
          SELECT * FROM cust;

CREATE RULE
postgres # CREATE TRIGGER cust_view_modify_after_trig
          AFTER INSERT OR UPDATE OR DELETE ON cust
          FOR EACH ROW
          EXECUTE PROCEDURE cust_view_modify_trig_
proc();
ERROR: "cust_view" is not a table
```

Хм!.. И что же это, если не таблица?

```
postgres # DROP TABLE cust_view;  
ERROR: "cust_view" is not a table  
HINT: Use DROP VIEW to remove a view  
postgres # DROP VIEW cust_view;  
DROP VIEW
```

Сработало! Да, вы создали таблицу, а потом применили к ней правило и таким образом *превратили таблицу в представление*. Теперь понятно, что невозможно применить триггеры к представлению, а правило `SELECT` – к таблице, если нужно, чтобы она оставалась таблицей, а не представлением. То есть этот способ работать не будет. Видимо, следует признать, что если потребуется подгрузить данные в таблицу, то нужно работать непосредственно с таблицей, а не с представлением. В PostgreSQL 9.1 планируется поддержка триггеров `INSTEAD OF` в представлениях, то есть обновление представлений будет полностью соответствовать стандарту SQL.



ГЛАВА 8.

Мониторинг и диагностика

Рассматриваемые темы:

- Подключен ли пользователь
- Что выполняет пользователь
- Активен или заблокирован запрос
- Почему запрос заблокирован
- Как прервать сессию
- Принятие решения о сомнительной подготовленной транзакции
- Использует ли кто-либо определенную таблицу
- Когда в последний раз использовали таблицу
- Сколько места на диске будут занимать временные данные
- Почему запросы выполняются медленно
- Обнаружение ошибок и сообщение о них
- Подведение итога записей в журнале об ошибках за день

Введение

В этой главе приводятся решения распространенных задач мониторинга и диагностики, которые могут понадобиться вам в работе с вашей базой данных. Здесь вы найдете ответы на вопросы, которые часто возникают при работе с PostgreSQL.

Важность мониторинга

Базы данных не являются изолированными объектами. Они «живут» в компьютере, используют процессор, оперативную память и подсистемы диска. Пользователи подключаются к базе данных по сети. В зависимости от настроек сетевые ресурсы могут понадобиться базам данных для работы, например для аутентификации пользователей, для использования дисков, которые были подключены по сети (хотя лучше так не делать), или для удаленного вызова функций при доступе к другим базам данных.

Это означает, что недостаточно проводить мониторинг только базы данных. Как минимум необходимо отслеживать все, что имеет отношение к ее использованию, задавая себе следующие вопросы:

- Доступен ли узел, на котором находится база данных? Принимает ли он соединения?
- Какова ширина занятого канала? Не было ли сетевых прерываний и оборванных соединений?
- Достаточно ли имеющегося объема оперативной памяти для выполнения основных задач? Сколько памяти доступно?
- Достаточно ли свободного места на диске? Когда оно исчерпается?
- Справляется ли дисковая подсистема? Какую дополнительную нагрузку она может выдержать?
- Выдерживает ли нагрузку процессор? Насколько заняты процессоры (ядра) системы?
- Доступны ли при необходимости другие сетевые сервисы, от которых зависит доступ к базе данных? (Например, если для аутентификации вы используете Kerberos, за ним тоже нужно следить.)
- Сколько раз за время работы происходило переключение контекста?

И, как правило, интересна история этих данных. Что было вчера? Неделью назад? Когда использование диска резко изменилось?

Если вы работаете с большой базой данных, то, скорее всего, у вас уже есть какие-либо средства мониторинга состояния узлов и сети.

Мониторинг имеет два аспекта: сбор данных по истории использования системы, показывающих, как она изменялась, и оповещение о критических ошибках. Инструменты, основанные на RRDtool (Round Robin Database tool), такие как Cacti и Munin, часто используются для сбора данных по истории использования серверов во всех аспектах и представления полученной информации в удобной для изучения графической форме. Сопоставление временных графиков по нескольким характеристикам в одинаковом масштабе может существенно прояснить поведение системы.

Другой аспект мониторинга – получение тревожных сигналов, свидетельствующих о каких-то серьезных сбоях и необходимости немедленного вмешательства.

Для оповещения часто используют Nagios.

Кроме того, разумеется, есть протокол SNMP (Simple Network Management Protocol), который поддерживается большим количеством

коммерческих приложений для мониторинга. Базовая поддержка мониторинга PostgreSQL через SNMP предусмотрена в pgsnmpd, доступной по адресу: <http://pgsnmpd.projects.postgresql.org/>

Передача данных о PostgreSQL инструментам мониторинга

Историю событий удобнее всего использовать, когда вся она доступна из одного источника и в едином временном масштабе. Большинство систем мониторинга имеют модульную архитектуру, и добавление новых типов входных данных означает установку новых модулей. Иногда приходится самостоятельно писать их; впрочем, написать модуль для инструмента наподобие Cacti нетрудно – надо лишь создать скрипт, выводящий отслеживаемые характеристики в простом текстовом формате.

Полезно иметь графическое представление для таких характеристик, как число соединений, использование диска, количество запросов, количество WAL-файлов, большей части характеристик из `pg_stat_user_tables` и `pg_stat_user_indexes` и т. д.

Выше представлен снимок экрана Cacti, на котором видны данные об использовании процессора, диска и сетевого соединения, пула соединений `pgbouncer`, нескольких соединений с клиентами PostgreSQL. Как видите, между графиками легко проследить связь.

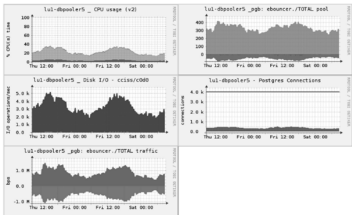


Рис. 8.1

Универсальный инструмент, который можно использовать и в Cacti, и в Nagios, – скрипт `check_postgres`, доступный по адресу: http://bucardo.org/check_postgres/check_postgres.pl.html

В нем предусмотрено оповещение о многих вещах, которые следует держать под контролем в PostgreSQL. Похожий инструмент для Nagios доступен по адресу: <http://pgfoundry.org/projects/nagiosplugins/>

В репозитории Munin есть несколько модулей PostgreSQL по следующему адресу: <http://exchange.munin-monitoring.org/plugins/search?keyword=postgres>

Дополнительная информация об основных инструментах мониторинга

Настройка инструментов мониторинга – отдельная тема, обсуждение которой не входит в задачу данной книги. Про каждый из вышеназванных инструментов написаны отдельные тома. Начальные сведения по настройке, равно как и сами инструменты, можно найти по следующим адресам:

- RRDtool (<http://www.mrtg.org/rrdtool/>);
- Cacti (<http://www.cacti.net/>);
- Munin (<http://munin-monitoring.org/>);
- Nagios (<http://www.nagios.org/>).

Наблюдение в реальном времени с помощью pgAdmin

С помощью `pgAdmin` можно быстро посмотреть, что происходит с базой данных. Для этого подключитесь к базе данных, выберите пункт меню **Tools → Server Status**. Откроется окно – примерно такое, как показано на следующей странице. В нем отображаются блокировки и действующие транзакции:

Подключен ли пользователь

В этом разделе будет показано, как узнать, подключен ли определенный пользователь к базе данных в настоящий момент.

Подготовка

Убедитесь, что вы зарегистрировались в системе как суперпользователь.

PID	Database	Relation	User	XID	TX	Mode	Granted	Start	Query
13765	rep0_master	24957	hannu	9/13126		RowExclusive...	Yes	2010-09-29 ...	UPDATE pgbench_branches SET bk...
13765	rep0_master	24957	hannu	9/13126		ShareLock	No	2010-09-29 ...	UPDATE pgbench_branches SET bk...
13765	rep0_master	24953	hannu	9/13126		AccessShare...	Yes	2010-09-29 ...	UPDATE pgbench_branches SET bk...
13765	rep0_master	24953	hannu	9/13126		RowExclusive...	Yes	2010-09-29 ...	UPDATE pgbench_branches SET bk...
13765	rep0_master	24971	hannu	9/13126		AccessShare...	Yes	2010-09-29 ...	UPDATE pgbench_branches SET bk...
13765	rep0_master	24971	hannu	9/13126		RowExclusive...	Yes	2010-09-29 ...	UPDATE pgbench_branches SET bk...
13765	rep0_master	24983	hannu	9/13126		RowExclusive...	Yes	2010-09-29 ...	UPDATE pgbench_branches SET bk...
13766	rep0_master	24986	hannu	11/1102		AccessShare...	Yes	2010-09-29 ...	UPDATE pgbench_tellers SET bala...
13766	rep0_master	24986	hannu	11/1102		AccessShare...	Yes	2010-09-29 ...	UPDATE pgbench_tellers SET bala...
13766	rep0_master	24971	hannu	11/1102		RowExclusive...	Yes	2010-09-29 ...	UPDATE pgbench_tellers SET bala...
13766	rep0_master	24983	hannu	11/1102		RowExclusive...	Yes	2010-09-29 ...	UPDATE pgbench_tellers SET bala...
13766	rep0_master	24986	hannu	11/1102		RowExclusive...	Yes	2010-09-29 ...	UPDATE pgbench_tellers SET bala...
13766	rep0_master	24986	hannu	11/1102		ExclusiveLock	Yes	2010-09-29 ...	UPDATE pgbench_tellers SET bala...

User	Client	Client start	Query start	TX start	Blocked & Query
hannu	127.0.0.1:33789	2010-09-29 10:39:34...			<IDLE>
hannu	local pipe	2010-09-29 10:48:29...			<IDLE>
londiste_picker	127.0.0.1:52144	2010-09-29 13:29:25...			<IDLE>
londiste_replicator	127.0.0.1:52161	2010-09-29 13:29:12...			<IDLE>
londiste_replicator	127.0.0.1:52168	2010-09-29 13:29:31...			<IDLE>
londiste_replicator	127.0.0.1:52294	2010-09-29 13:31:12...			<IDLE>
londiste_replicator	127.0.0.1:52320	2010-09-29 13:33:12...			<IDLE>
londiste_picker	127.0.0.1:34233	2010-09-29 13:39:34...			<IDLE>
hannu	local pipe	2010-09-29 13:42:02...	2010-09-29 ...	2010-09-29 ...	UPDATE pgbench_branches SET bala...
hannu	local pipe	2010-09-29 13:42:02...	2010-09-29 ...	2010-09-29 ...	UPDATE pgbench_tellers SET bala...
hannu	local pipe	2010-09-29 13:42:02...	2010-09-29 ...	2010-09-29 ...	UPDATE pgbench_tellers SET bala...
hannu	local pipe	2010-09-29 13:42:02...	2010-09-29 ...	2010-09-29 ...	UPDATE pgbench_branches SET bala...
hannu	local pipe	2010-09-29 13:42:02...	2010-09-29 ...	2010-09-29 ...	END
hannu	local pipe	2010-09-29 13:42:02...	2010-09-29 ...	2010-09-29 ...	UPDATE pgbench_branches SET bala...
hannu	local pipe	2010-09-29 13:42:02...	2010-09-29 ...	2010-09-29 ...	UPDATE pgbench_branches SET bala...

Рис. 8.2

Как это сделать

Выполните следующий запрос к базе, чтобы проверить, подключен ли пользователь bob:

```
SELECT datname FROM pg_stat_activity WHERE username = 'bob';
```

Если по этому запросу возвращена хоть одна строка, то пользователь bob подключен к базе. Возвращаемое значение – имя базы данных, к которой он подключен.

Как это работает

Системное представление PostgreSQL `pg_stat_activity` отслеживает всех активных пользователей, выполняемые запросы, подключенных пользователей и время их подключения, а также время запуска текущей транзакции и текущего запроса.

И еще

В представлении `pg_stat_activity` содержится не только имя пользователя. Допустим, пользователь спрашивает:

«Что если я хочу знать, подключен ли тот компьютер?»

Часто несколько различных процессов могут подключаться от имени одного пользователя. В таком случае возникает вопрос, осуществляется ли подключение из определенного места.

Эту информацию также можно получить из `pg_stat_activity`, поскольку это представление содержит IP и порт подключившегося клиента. Номер порта надо знать, если у клиента несколько подключений с одного компьютера и требуется выяснить, какой процесс и к какой базе подключен. Выполните следующий запрос:

```
SELECT datname,username,client_addr,client_port FROM pg_stat_activity ;
```

Значения в столбцах `client_addr` и `client_port` позволят узнать, какой компьютер и даже какой процесс подключен к этой базе.

Что выполняет пользователь

Ниже мы покажем, как узнать, какой запрос выполняется в данный момент.

Подготовка

Убедитесь, что вы зарегистрировались как суперпользователь или от имени пользователя, которого хотите проверить.

Убедитесь, что задан параметр `track_activities = on`.

Это можно сделать либо отредактировав файл `postgresql.conf`, либо выполнив от имени суперпользователя следующую команду SQL:

```
SET track_activities = on;
```

Как это сделать

Чтобы увидеть, что делают все подключенные пользователи, выполните следующее:

```
SELECT datname,username,current_query FROM pg_stat_activity ;
```

В системе, обслуживающей многих пользователей, можно заметить, что большинство из них выполняют странный запрос `<IDLE>`. Это обозначение состояния, в котором PostgreSQL не выполняет никаких запросов, ожидая от пользователя ввода команд.

Чтобы получить информацию только об активных запросах, исключите бездействующие:

```
SELECT datname,username,current_query  
FROM pg_stat_activity  
WHERE current_query <> '<IDLE>' ;
```


Как это работает

Когда задан параметр `track_activities = on`, PostgreSQL собирает данные обо всех выполняемых запросах. Пользователи, имеющие соответствующие права, могут получить эти данные из системного представления `pg_stat_activities`.

Представление `pg_stat_activities` использует системную функцию `pg_stat_get_activity(procpid int)`, которую можно непосредственно вызывать с ID процесса в качестве аргумента для наблюдения за активностью конкретного пользователя. Вызванная с аргументом `NULL`, функция возвращает информацию обо всех пользователях.

И еще

Иногда нужна информация не обо всех выполняемых запросах, а только о некоторых. Или вы не хотите подключаться к базе данных просто для того, чтобы посмотреть, с какими запросами к ней обращаются.

Как увидеть запросы, выполняемые лишь в течение нескольких миллисекунд

Большинство запросов на современных системах с использованием OLTP (Online Transaction Processing, транзакционная система [реального времени]) выполняется за несколько миллисекунд, и трудно заметить каждый из них, просто проверяя таблицу `pg_stat_activity`.

Чтобы увидеть эти запросы в процессе выполнения, их пришлось бы затормозить. Но это нам не подходит!

Для PostgreSQL 9.0 есть модуль от сторонних разработчиков `pg_stat_statements`, который собирает статистику по выполняемым запросам в реальном времени. Документацию можно найти по адресу: <http://www.postgresql.org/docs/9.0/interactive/pgstatstatements.html>

В обоих случаях можно написать скрипт с запросом и использовать `select pg_sleep(5)`, чтобы состояние `<wait>` достигалось автоматически. Ожидание короче секунды можно получить, используя дробные числа, например `pg_sleep(0.5)` для полусекундной задержки.

Чтобы собрать информацию о запросах, которые вы только что заставили логироваться, выполните команду `tail -f /var/log/postgresql/postgresql-9.0-main.log > account_queries.log` в одном окне, а в другом проделайте вышеописанный трюк с блокировками и просто нажмите **Ctrl+C**, чтобы завершить процесс `tail -f`.

Теперь у вас намного меньший лог в файле `account_queries.log`.

Как отслеживать самые долгие запросы

Еще одна вещь, которая может заинтересовать, – долго выполняемые запросы. Вот как можно получить список запросов, отсортированный по времени выполнения:

```
select
    current_timestamp - query_start as runtime,
    datname,
    username,
    current_query
from pg_stat_activity
where current_query != '<IDLE>'
order by 1 desc;
```

Будет возвращен упорядоченный список выполняемых в настоящий момент запросов, начиная с тех, которые обрабатываются дольше всего (первое поле: `order by 1 desc`). Для нагруженной системы имеет смысл ограничить список несколькими первыми строками (добавьте в конце `LIMIT 10`) или строками для запросов, выполняемых дольше определенного времени (так, для запросов, выполняемых дольше одной минуты, добавьте `'current_timestamp - query_start 1'` в оператор `WHERE`).

Слежение за запросами с помощью ps

Если вы хотите, чтобы выполняемые процессы отображались в именах процессов, установите параметр

```
update_process_title = on
```

Хотя `ps` или `top` – не лучший способ слежения за запросами к базе данных, в некоторых случаях это может иметь смысл.

Смотрите также

Документация по PostgreSQL, охватывающая смежные темы, доступна по адресу: <http://www.postgresql.org/docs/9.0/interactive/runtime-config-statistics.html>.

Активен или заблокирован запрос

Ниже будет показано, как определить, выполняется запрос или ждет завершения другого запроса.

Подготовка

Снова зарегистрируйтесь в системе как суперпользователь.

Как это сделать

Выполните следующий запрос:

```
SELECT datname,username,current_query
FROM pg_stat_activity
WHERE waiting = true;
```

Вы получите список запросов, ожидающих других обработчиков.

Как это работает

Предыдущий запрос отфильтровывает только ожидающие запросы.

И еще

Вышесказанное требует некоторых пояснений.

Нет необходимости в «= true»

Так как столбец содержит значение типа `boolean`, можно смело пропустить `= true` в запросе. Достаточно написать:

```
SELECT datname,username,current_query
FROM pg_stat_activity
WHERE waiting;
```

Будут определены только запросы, ожидающие разблокирования.

Поле `pg_stat_activity.waiting` показывает, ожидает ли запрос освобождения внутренней блокировки PostgreSQL.

Хотя это основная причина ожидания при использовании чистого SQL, может оказаться так, что вы будете иметь дело с кодом, написанным на одном из встроенных языков PostgreSQL, который в некоторых ситуациях будет ожидать какого-либо системного ресурса, например HTTP-ответа, окончания записи в файл или просто таймера.

Пример:

Напишем простую функцию на PL/PythonU (буква **U** означает «не доверенный»; только суперпользователи могут создавать функции на этом языке):

```
create or replace function wait(seconds float)
returns void as $$
import time;
time.sleep(seconds)
```

```
$$  
language plpythonu;
```

Когда вы выполняете следующую функцию:

```
db=# select wait(10);  
-- здесь останавливаемся и ждем 10 секунд  
wait  
-----  
(1 row)
```

то она в представлении `pg_stat_activity` не будет показана как ожидающая, несмотря на то что запрос фактически «блокирован» таймером.

Почему запрос блокирован

Если вы обнаружили, что некий запрос блокирован, надо выяснить, кто или что блокирует его.

Подготовка

Как и в других случаях, используйте учетную запись суперпользователя для выполнения запросов.

Как это сделать

Выполните следующий запрос:

```
SELECT  
    w.current_query as waiting_query,  
    w.procpid as w_pid,  
    w.username as w_user,  
    l.current_query as locking_query,  
    l.procpid as l_pid,  
    l.username as l_user,  
    t.schemaname || '.' || t.relname as tablename  
from pg_stat_activity w  
join pg_locks l1 on w.procpid = l1.pid and not l1.granted  
join pg_locks l2 on l1.relation = l2.relation and  
l2.granted  
join pg_stat_activity l on l2.pid = l.procpid  
join pg_stat_user_tables t on l1.relation = t.relid  
where w.waiting;
```

Он возвратит ID процесса, имя пользователя и выполняемый запрос для обоих обработчиков – блокируемого и блокирующего, а также схему и имя таблицы, вызвавшей блокировку.

Как это работает

Этот запрос сначала выбирает все ожидающие запросы (`where w.waiting`), затем получает блокировки, которых он ждет (`join pg_locks l1 on w.procpid = l1.pid and not l1.granted`), потом находит установленную блокировку в той же таблице (`join pg_locks l2 on l1.relation = l2.relation and l2.granted`) и, наконец, обращается к строке в `pg_stat_activity`, соответствующей установленной блокировке. Кроме того, он получает по идентификатору (`relid`) таблицы ее полное имя, используя системное представление `pg_stat_user_tables`.

Как прервать сессию

Иногда для того, чтобы система в целом могла продолжать работать, необходимо принудительно завершить сеансы обращения к базе данных, которые ведут себя неправильно.

Подготовка

Задачу могут выполнять только суперпользователи. Зарегистрируйтесь в системе как суперпользователь.

Как это сделать

Найдите пользователя, сеанс которого хотите завершить, и вызовите функцию

```
pg_terminate_backend(processid)
```

которая, собственно, и выполнит задачу.

Как это работает

Когда пользователь вызывает функцию `pg_terminate_backend(processid)`, она посылает сигнал `SIGQUIT` процессу, ID которого указан в качестве аргумента, после того как было подтверждено, что указанный процесс является сеансом PostgreSQL.

Сеанс, получивший сигнал, прекращает все свои действия и завершается контролируемым образом.

Пользователь, сеанс которого принудительно завершен, теряет соединение с базой. В зависимости от того как был запущен сеанс, он может перезапуститься без выдачи сообщений либо выдать пользователю сообщение об ошибке.

И еще

Принудительное завершение сеанса – не всегда лучшее решение, поэтому рассмотрите другие варианты.

Попытайтесь отменить запрос

Возможно, предпочтительнее использовать менее жесткий вариант, `pg_cancel_backend(processid)`.

Разница состоит в том, что `pg_cancel_backend()` отменяет выполняемый запрос, в то время как `pg_terminate_backend()` принудительно завершает сеанс/процесс PostgreSQL целиком.

Если сеанс не завершается

Если `pg_terminate_backend(processid)` не может принудительно завершить сеанс и необходимо, чтобы база данных продолжала обслуживать запросы, остается послать сигнал `SIGKILL` (сигнал с номером 9) процессу.

Это можно сделать только через командную строку от имени системного администратора или пользователя `postgres` на том компьютере, где установлена база данных, с помощью команды:

```
kill -9 <backendpid>
```

которая без промедления завершает сеанс, не позволяя закончить его корректно и заставляя таким образом `postmaster` завершить другие сеансы и перезапустить весь кластер.

Поэтому не имеет значения, какой именно из сеансов получает `SIGKILL`.

Учтите, что при значении параметра `synchronous_commit` равном `off` вы можете потерять некоторые предположительно завершенные транзакции, если «убиваете» (`kill -9`) сеанс.

Итак, `kill -9` является крайним средством на случай, если ничто другое не помогает, и не годится для повседневного использования.

Используйте ограничение времени выполнения запроса

Зачастую выполнение запроса по истечении промежутка времени `x` теряет смысл. Например, веб-интерфейс через десять секунд перестает ждать результат запроса и возвращает пользователю некоторое стандартное значение.

В таком случае полезно установить параметр `statement_timeout` = 15 сек либо в файле `postgresql.conf`, либо в индивидуальных

установках для пользователей или баз, чтобы слишком долгие запросы не потребляли ресурсы понапрасну и не мешали выполнению других запросов.

Запросы, завершенные по тайм-ауту, оставляют в логах сообщения примерно следующего содержания:

```
hannu=# set statement_timeout = '3 s';
SET
hannu=# select wait(10);
ERROR: canceling statement due to statement timeout
```

В более ранних версиях PostgreSQL такие запросы оставляли менее осмысленные сообщения `query canceled due to user request` («запрос отменен пользователем»).

Завершение запросов `Idle in transaction`

Иногда пользователь начинает транзакцию и, выполнив несколько запросов, оставляет ее незавершенной. При этом некоторые системные ресурсы остаются в состоянии, препятствующем процессам обслуживания базы данных, или же последствия оказываются более серьезными, например блокируется доступ к базе других пользователей.

Следующий запрос завершит все сеансы, в которых открыта транзакция, но не выполнялось никаких действий за последние 10 минут:

```
select pg_terminate_backend(procpid)
from pg_stat_activity
where current_query = '<IDLE> in transaction'
and current_timestamp - query_start > '10 min';
```

Можно также назначить автоматическое выполнение такого запроса ежеминутно, на случай если какой-либо веб-интерфейс завершил работу, не закрыв сеанс, или невнимательный администратор оставил открытым соединение `psql`, или потеря клиентом соединения в плохо сконфигурированной сети осталась незамеченной сервером.

Сеанс можно завершить из командной строки

Еще один способ завершить сеанс в Unix/Linux – команда `kill N`, которая посылает сигнал процессу `N` в системе, в которой запущен процесс. Чтобы посылать процессам сигналы, нужно быть зарегистрированным в системе как `root` или как пользователь, запускающий сеансы базы данных (обычно `postgres`).

Принятие решения о сомнительной подготовленной транзакции

При использовании 2PC (*two phase commit*, «двухфазный коммит») можно столкнуться с ситуацией, когда что-то заблокировано, но неизвестен блокирующий сеанс.

Например:

```
db=# select t.schemaname || '.' || t.relname as tablename,  
db=# l.pid, l.granted  
db=# from pg_locks l join pg_stat_user_tables t  
db=# on l.relation = t.relid;  
tablename | pid | granted  
-----+-----+-----  
db.x      |    | t  
db.x      | 27289 | f  
(2 rows)
```

Здесь мы видим блокировку на таблице `db.x`, с которой не ассоциирован ни один процесс.

Подготовка

Смотрите раздел «Удаление ранее подготовленных транзакций» в девятой главе.

Использует ли кто-либо определенную таблицу

Если неясно, использует ли кто-то определенную таблицу или это «пережиток прошлого», без пользы занимающий место на диске, воспользуйтесь следующими рекомендациями.

Подготовка

Убедитесь, что вы зарегистрированы как суперпользователь или по крайней мере обладаете полными правами в отношении данной таблицы.

Как это сделать

Чтобы выяснить, используется ли таблица в данный момент, выполните следующее:

```
create temp table tmp_stat_user_tables as select * from  
pg_stat_user_tables;
```


Потом немного подождите и посмотрите, что изменилось.

```
select * from pg_stat_user_tables n
join tmp_stat_user_tables t
on n.relid=t.relid
and (n.seq_scan,n.idx_scan,n.n_tup_ins,n.n_tup_upd,n.n_tup_del) <>
(t.seq_scan,t.idx_scan,t.n_tup_ins,t.n_tup_upd,t.n_tup_del);
```

Как это работает

Таблица `pg_stat_user_tables` – представление, которое показывает текущую статистику использования таблиц.

Чтобы определить, используется ли таблица, ищите изменения в показателях ее использования.

Предыдущий запрос выбирает все таблицы, показатели использования которых как-либо изменились.

И еще

На скорую руку

Если в накопленной PostgreSQL статистике нет необходимости, можно сбросить статистику по таблицам:

```
select pg_stat_reset();
```

Вся статистика обнуляется; после этого используемые таблицы можно определить, просто находя ненулевые показатели использования.

Можно предварительно сделать резервную копию таблицы статистики:

```
create table backup_stat_user_tables as
select current_timestamp as snaptime, *
from pg_stat_user_tables;
```

Ежедневный сбор статистики использования

Часто для решения проблем с производительностью полезно иметь под рукой статистику использования таблиц; кроме того, она помогает понять характер использования базы данных.

Для этой цели можно собирать данные по использованию регулярно – ежедневно или даже чаще – с помощью `cron` или планировщика PostgreSQL, такого как `pg_agent`.

Следующий запрос добавляет снимок текущей статистики использования с временными пометками в таблицу, созданную ранее:

```
insert into backup_stat_user_tables
select current_timestamp as snaptime, *
from pg_stat_user_tables;
```

Когда в последний раз использовали таблицу

Если ясно, что таблица не используется сейчас, возникает следующий вопрос: когда ее использовали в последний раз?

Подготовка

Получите доступ к базе данных как суперпользователь либо доступ к системе, где установлена база, как пользователь `postgres`.

Как это сделать

PostgreSQL не имеет встроенных средств сбора информации о последнем обращении к таблицам, поэтому придется найти другие средства.

Если в расписании `cron` запланировано задание по сбору статистики использования (см. предыдущую главу), то определить дату последнего изменения с помощью запроса SQL относительно легко.

В противном случае остаются два способа, ни один из которых, впрочем, не дает столь же достоверного ответа.

Можно либо просмотреть временные отметки файлов, в которых хранятся данные, либо поискать в системных столбцах `xmin` и `xmax` ID транзакции, которая изменила данные в таблице.

Даты в атрибутах файла

Для определения имени одного из нескольких файлов, содержащих данные таблицы, надо выполнить нижеописанные действия.

Вот пример функции на языке PL/PythonU, которая выдает атрибуты файлов, содержащих таблицу. Для того чтобы она работала, должен быть установлен PL/PythonU. Если его нет, прибегните к следующему методу:

```
CREATE LANGUAGE plpythonu;
```

чтобы использовать транслятор для работы с базой данных. Понимается, что PL/PythonU поддерживается в системе, где установлена база данных.

Для начала создадим тип данных, которому будет принадлежать возвращаемое функцией значение; далее следует собственно функция:

```
CREATE TYPE fileinfo AS (
    filename text,
    filesize bigint,
    ctime abstime,
    mtime abstime,
    atime abstime
);

CREATE OR REPLACE FUNCTION table_file_info(schemaname text,
tablename
text)
RETURNS SETOF fileinfo
AS $$
import datetime, glob, os
db_info = plpy.execute("""
select datname as database_name,
       current_setting('data_directory') || '/base/' ||
db.oid as
data_directory
  from pg_database db
 where datname = current_database()
""")
#return db_info[0]['data_directory']
table_info_plan = plpy.prepare("""
select nspname as schemaname,
       relname as tablename,
       relfilenode as filename
  from pg_class c
 join pg_namespace ns on c.relnamespace=ns.oid
 where nspname = $1
       and relname = $2;
""", ['text', 'text'])
table_info = plpy.execute(table_info_plan, [schemaname,
tablename])
filemask = '%s/%s' % (db_info[0]['data_directory'], table_
info[0]
['filename'])
res = []
for filename in glob.glob(filemask):
    fstat = os.stat(filename)
    res.append((
        filename,
        fstat.st_size,
        datetime.datetime.fromtimestamp(fstat.st_ctime).
isoformat(),
        datetime.datetime.fromtimestamp(fstat.st_mtime).
isoformat(),
        datetime.datetime.fromtimestamp(fstat.st_atime).
isoformat())
```

```
    ))  
    return res  
    $$ LANGUAGE plpythonu;
```

Теперь для получения времени последнего изменения таблицы и времени последнего доступа к ней используйте следующий запрос:

```
select  
    max(mtime) as latest_mod,  
    max(atime) as latest_read  
from table_file_info(<schemaname>, <tablename>);
```

Как это работает

Функция `table_file_info(schemaname, tablename)` возвращает время создания и изменения файлов, используемых PostgreSQL для хранения данных таблицы, и время доступа к ним.

Последний запрос использует эти данные для определения времени последнего обращения PostgreSQL к какому-либо из этих файлов. Такой способ не очень надежен, поскольку последнее обращение к файлу не обязательно было связано с использованием таблицы; тем не менее он позволяет оценить самое позднее время, когда могло произойти последнее обращение к таблице.

Консольный доступ дает возможность выполнить вышеописанные действия вручную, если PL/PythonU по каким-либо причинам нежелательно использовать.

Также можно получить искомые данные при помощи встроенных функций `pg_ls_dir(dirname text)` и `pg_stat_file(filename text)`. Например, следующий запрос:

```
select pg_ls_dir, (select modification from pg_stat_  
file(pg_ls_dir))  
as modtime from pg_ls_dir('.');
```

выдает список всех файлов и поддиректорий в директории данных PostgreSQL.

И еще

В будущих версиях PostgreSQL могут появиться встроенные средства определения времени последнего обращения к данным.

Недавно прошла дискуссия по поводу добавления времени последнего использования таблицы к хранимым PostgreSQL данным о таблицах, и возможно, что ответить на вопрос «Когда состоялось последнее обращение к таблице?» будет гораздо легче в следующей версии PostgreSQL.

Сколько места на диске будут занимать временные данные

Кроме обычных постоянных таблиц можно использовать временные таблицы.

Также PostgreSQL может использовать временные файлы при обработке запросов, если недостаточно оперативной памяти для размещения всех необходимых данных.

Посмотрим, как определить объем используемых временных таблиц и файлов.

Подготовка

Эту задачу, как и вышеописанные, можно решить с помощью любого недоверенного встроенного языка либо прямо в системе, где установлена база данных. Язык должен быть недоверенным, так как доверенные работают в «песочнице» без прямого доступа к системе.

Как это сделать

Для начала проверьте, определены ли в вашей базе данных специальные табличные пространства, как в следующем примере:

```
select current_setting('temp_tablespace');
```

Когда temp_tablespace содержит одно или несколько табличных пространств

В этом случае задача упрощается, поскольку все временные файлы – для временных таблиц и для обработки запросов – находятся внутри директорий этих табличных пространств. Просто найдите соответствующие директории в pg_tablespace, как в следующем примере:

```
select spcname,spclocation from pg_tablespace;
```

Консольная команда `du` покажет используемое временными файлами пространство. Вот пример:

```
db=# select current_setting('temp_tablespace');
current_setting
-----
temp1, temp2
(1 row)
db=#
db=# select spcname,spclocation from pg_tablespace where
spcname in
```

```

('temp1', 'temp2');
  spcname | spclocation
-----+-----
  temp1   | /test/pg_tmp1
  temp2   | /test/pg_tmp2
(2 rows)
db=# \q
user@host:~$
sudo du -s /test/pg_tmp1 /test/pg_tmp2
102136
/test/pg_tmp1
35144      /test/pg_tmp2

```

Ввиду того что объем дискового пространства, занятого временными файлами в активной системе, может существенно изменяться, имеет смысл повторить команду `du -s` несколько раз, чтобы получить лучшее представление о том, как меняется использование диска.

Когда в `temp_tablespaces` пусто

Если опция `temp_tablespaces` не задана, временные таблицы хранятся среди обычных таблиц, а временные файлы при обработке запросов создаются в директории `pgsql_tmp_directory` внутри главной директории базы данных.

Найдите домашнюю директорию кластера следующим образом:

```
select current_setting('data_directory') || '/base/pgsql_tmp'
```

Занимаемый этой директорией объем равен общему объему используемых в данный момент временных файлов для обработки запросов.

Общий объем временных файлов, используемых базой данных, можно найти следующим образом:

```
select sum(pg_total_relation_size(re lid))
from pg_stat_all_tables
where schemaname like 'pg_%temp%';
```

Как это работает

Поскольку все временные таблицы и другие временные данные хранятся на диске в файлах, вы можете определить расположение этих файлов с помощью внутренних таблиц PostgreSQL, а затем узнать размер файлов.

И еще

Все сказанное выше о временных таблицах верно, однако это еще не все, что следует знать.

Как выяснить, использует ли кто-либо временные файлы

Так как хранение временных файлов по сравнению с обычными таблицами требует меньших мер предосторожности (на самом деле это их преимущество, поскольку делает обращение к ним более быстрым), то иногда после аварийной перезагрузки системы некоторые временные файлы остаются на диске и могут занимать значительную часть дискового пространства.

Для зачистки можно выгрузить сервер PostgreSQL и удалить все файлы из директории `pgsql_temp`.

Логирование использования временных файлов

Если задан параметр `log_temp_files`, то создание временных файлов, размер которых в килобайтах превышает значение, присвоенное этому параметру, отображается в стандартном лог PostgreSQL.

Почему запросы выполняются медленно

Выполнение запросов, ранее занимавшее десятки миллисекунд, внезапно затягивается до нескольких секунд.

Запрос для итогового отчета, на который прежде уходило несколько секунд, теперь требует получаса.

Есть несколько способов выяснить, что замедляет выполнение запросов.

Подготовка

На любые вопросы типа «Почему сегодня все не так, как на прошлой неделе?» проще ответить, если в системе собирается статистика.

Такие программы, как `Cacti` или `Munin`, предназначенные для мониторинга основных характеристик сервера (использование процессора и оперативной памяти, дисков, сетевых соединений, средняя загрузка системы), помогают узнать, что изменилось в последнее время, и соотнести данные об изменениях с наблюдаемой производительностью некоторых операций в базе данных.

Также история статистических данных из таблиц `pg_stat_*`, собираемая ежедневно, ежечасно, даже каждые пять минут, если позволяют

ресурсы, очень полезна для обнаружения вероятных причин внезапных изменений или постепенного снижения производительности.

Еще лучше применять оба способа сбора статистики. Если не применяется ни один, то вопрос ставится иначе: «Почему этот запрос выполняется медленно?»

Но не отчаивайтесь – есть способы восстановить производительность.

Как это сделать

Сначала проанализируйте базу данных следующим образом:

```
db_01=# analyse;  
ANALYZE  
Time: 6231.313 ms  
db_01=#
```

Это первое, что следует сделать; такой вариант обходится дешево и рекомендуется для частого использования.

Если в результате производительность запросов восстановилась или хотя бы заметно повысилась, это означает, что `autovacuum` плохо делает свою работу, и следует проверить, в чем причина.

Как это работает

`ANALYZE` обновляет статистику по объемам и распределению данных во всех таблицах. Если размер таблицы значительно изменился, а статистика не обновлялась, то оптимизатор PostgreSQL, основанный на статистике, может выбрать неверную стратегию. При выполнении команды `ANALYZE` вручную обновляется статистика по всем таблицам.

И еще

Есть еще несколько распространенных проблем.

Возвращают ли запросы значительно больше данных, чем прежде?

Если первоначальное тестирование проводилось на почти пустых таблицах, вполне возможно, что запрашивается намного больше данных, чем необходимо.

Пример: если выбрать все данные пользователей, а показать лишь для первых десяти, то такой запрос выполняется очень быстро, когда одному пользователю соответствует 10 или 50 записей, но гораздо медленнее, когда этих записей 50 000.

Старайтесь включать в запросы лишь необходимые данные; используйте оператор `limit` для ограничения возвращаемых приложений данных (и чтобы дать оптимизатору возможность выбрать стратегию с меньшим объемом обрабатываемых данных, а также, возможно, меньшими начальными затратами).

Выполняются ли запросы так же медленно в отсутствие других запросов?

Если это возможно, сравните время выполнения запросов, когда других запросов нет или очень мало.

Если при низкой загрузке базы данных запросы выполняются быстрее, вероятно, что система перегружена (исчерпан ресурс процессора, оперативной памяти или дисковой подсистемы) и стратегия, оптимальная при малой нагрузке, показала свои слабые стороны. Возможно, стратегия и не была хороша, но высокое быстродействие современных компьютеров порой скрадывает недостатки.

```
db=# select count(*) from t;
count
-----
1000000
(1 row)
Time: 329.743 ms
```

Как видите, сканирование одного миллиона строк происходит за 0,3 с на ноутбуке недавнего года выпуска, если строки уже в кэше.

Но когда обрабатываются несколько запросов и при этом не помещаются одновременно в оперативную память, скорость выполнения данного запроса может снизиться на порядок или даже больше.

В десятой главе вы найдете некоторые советы по настройке производительности.

Выполняется ли запрос так же медленно во второй раз, как в первый?

Этот тест соотносится с предыдущим и позволяет определить, вызвано ли падение производительности недостатком оперативной памяти.

Если во второй раз запрос выполнялся быстрее, можно предположить, что проблема вызвана недостатком оперативной памяти.

Снова обратитесь к десятой главе.

Раздувание таблиц и индексов

Если система недостаточно аккуратно поддерживается, это со временем приводит к определенным последствиям, в том числе к

раздуванию таблиц. Согласно принципу работы MVCC, в таблице накапливается множество старых версий строк, если их своевременно не удалять.

Процесс раздувания может протекать по-разному, но в любом случае он основан на многочисленных добавлениях и изменениях строк, между тем как `autovacuum` не в состоянии удалять устаревшие кортежи. Если устаревшие версии строк удаляются, в конце таблицы могут находиться видимые строки, препятствуя сжатию файла. Известны случаи, когда таблица из одной строки разрасталась до нескольких гигабайтов.

Если кажется, что таблица раздулась, выполните следующее:

```
select pg_relation_size(relid) as
tablesize,schemaname,relname,n_live_
tup
from pg_stat_user_tables
where relname = <tablename>;
```

и проверьте, имеет ли смысл отношение `tablesize / n_live_tup`.

Например, таблица размером в десятки мегабайтов, содержащая лишь несколько строк, явно раздута.

Смотрите также

- Раздел «Использует ли кто-либо данную таблицу» (подраздел «Ежедневный сбор статистики использования»), где показан способ сбора информации об изменениях в таблице.
- Всю десятую главу.

Обнаружение ошибок и сообщение о них

Если вы обнаружили, что PostgreSQL делает не то, что следует, надо в этом разобраться.

Подготовка

Перед началом расследования разумно сделать полную резервную копию сервера PostgreSQL. Так вы сможете в любой момент начать исследования заново, не опасаясь, что наблюдаемые результаты вызваны не ошибкой, а изменениями, внесенными в процессе ее изучения.

Как это сделать

Попробуйте создать наиболее простой воспроизводимый сценарий, демонстрирующий ошибку в программе. Иногда упрощения позволяют обойти ошибку, но еще важнее, что они облегчают задачу тому, кто возьмется ее устранить. Практически невозможно устранить ошибку, если неизвестно, как ее вызвать, чтобы пронаблюдать.

Если дело касается обработки запросов, не мешает минималистичный дамп базы данных (результат выполнения `running_pg_dump`) вместе со скриптом SQL, вызывающим проявление ошибки.

Если повреждены данные, вероятно, следует предоставить файлы с этими данными (или с поврежденным участком данных) тому, кто сможет их изучить. Иногда такого специалиста можно найти в рассылке хакеров PostgreSQL, или где-то нанять, или даже справиться самостоятельно. Чем лучше проведена подготовительная работа, чем конкретнее сформулированы вопросы, тем вероятнее найдется быстрое решение проблемы.

Если подозревается повреждение данных и есть интерес разобраться в вопросе, почитайте о форматах данных по адресу <http://www.postgresql.org/docs/9.0/static/storage.html>, и исследуйте свои таблицы с данными с помощью пакета `pageinspect`, разработанного участниками сообщества PostgreSQL.

В журнале об ошибках обязательно следует указывать используемую версию PostgreSQL и операционную систему.

Детальное руководство можно найти в конце этой главы.

Как это работает

Оптимистичный сценарий таков:

- Пользователь отправляет хорошо подготовленный журнал об ошибках в рассылку хакеров PostgreSQL.
- В рассылке идет обсуждение, у пользователя могут запросить дополнительную информацию.
- Кто-то находит источник проблем и предлагает исправление.
- Исправление обсуждают в рассылке.
- Ошибку исправляют, изготавливают «заплатку» для текущей версии и включают ее в следующую версию.
- Иногда заплатку делают для предыдущих версий.

К сожалению, на каждом этапе может произойти сбой: допустим, никто не счел себя компетентным в данном вопросе, или у того, кто мог бы решить проблему, нет возможности ей заняться, или все понадеялись на кого-то, кто даже не читал рассылку.

Если на сообщение в рассылке не последовало реакции, через день или два попробуйте выяснить, почему так случилось.

Смотрите также

Официальные руководства PostgreSQL по составлению журналов об ошибках

Вы с высокой степенью вероятности сможете найти ответы на свои вопросы, если обратитесь по следующим адресам:

http://wiki.postgresql.org/wiki/Guide_to_reporting_problems

<http://wiki.postgresql.org/wiki/SlowQueryQuestions>

Подведение итога записей в журнале об ошибках за день

PostgreSQL может создавать гигабайты логов в день. Столь объемная информация может оказаться полезной для исследования проблемы, но не является необходимой для рутинного мониторинга состояния базы данных.

Подготовка

Убедитесь, что в PostgreSQL включена ежедневная ротация логов.

В стандартных настройках установлен параметр:

```
log_rotation_age = 1d
```

Воспользуйтесь программой обработки логов PostgreSQL. Здесь будет описано использование **pgFouine**.

В большинстве систем Linux программу pgFouine можно установить с помощью стандартного менеджера пакетов.

Настройте PostgreSQL так, чтобы логи создавались в формате, пригодном для обработки с помощью pgFouine. Выберите логирование в *syslog*. Используйте современную версию *syslogd* для базы данных под высокой нагрузкой.

Как это сделать

Запланируйте задание для `cron` на время через несколько минут после стандартной ротации логов. Время можно узнать по временным отметкам логов, уже прошедших ротацию: все они будут помечены схожим временем.

Выполните следующее:

```
pgfouine.php -file /var/log/postgresql/postgresql-9.0-  
main.log.1  
-reports n-mostfrequenterrors > errors.html
```

Можно также запланировать отправку отчетов по электронной почте администратору базы данных.

Как это работает

PgFouine уплотняет и сортирует сообщения об ошибках для удобного просмотра, создавая отчет, аккуратно отформатированный в HTML. В этом отчете легко увидеть часто встречающиеся ошибки.

Правильнее всего не оставлять ошибки в логах базы данных без внимания, если это возможно. Увидев ошибки в логах и отчетах, следует найти причину и исправить их.

Несмотря на соблазн оставить без последствий ошибки, не приносящие вреда, следует рассматривать их как возможное указание на более серьезные проблемы, которые впоследствии могут проявиться в не столь безобидной форме, вплоть до нарушения безопасности или повреждения данных.

И еще

Создание собственного обработчика логов

Если в логах число ошибок невелико, может быть достаточно использовать `grep` для их поиска:

```
user@dbhost: $ egrep "FATAL|ERROR" postgresql-9.0-main.log
```

Смотрите также

- Домашняя веб-страница pgFouine находится по адресу: <http://pgfouine.projects.postgresql.org/>
- Намного больше информации по настройке pgFouine по адресу: <http://pgfouine.projects.postgresql.org/tutorial.html>
- Похожий инструмент – PQA (Practical Query Analysis), доступный по адресу: <http://pqa.projects.postgresql.org/>

PQA написан на языке Ruby, и если вы хорошо знаете Ruby, то эта программа может быть наилучшим вариантом.

Для PostgreSQL 9.0 сообществом разработан модуль `pg_stat_statements`, который собирает статистику выполнения запросов в реальном времени. Документацию можно найти по адресу: <http://www.postgresql.org/docs/9.0/interactive/pgstatstatements.html>



ГЛАВА 9.

Регулярное обслуживание

Рассматриваемые темы:

- Управление автоматическим обслуживанием базы данных
- Предотвращение «автозаморозки» и повреждения страниц
- Предупреждение «свертывания» счетчика транзакций
- Удаление ранее подготовленных транзакций
- Для любителей временных таблиц
- Обнаружение и исправление раздутых таблиц и индексов
- Обслуживание индексов
- Обнаружение неиспользуемых индексов
- Безопасное удаление ненужных индексов
- Планирование обслуживания

Введение

Для PostgreSQL предпочтительно регулярное обслуживание, и в этой книге рассматривается планирование обслуживания.

Очевидно, читатель обращается к этой книге потому, что ищет готовые решения своих проблем. При этом проблемы, скорее всего, необходимо решить немедленно, а планированием заняться позднее. В таком случае приступим.

PostgreSQL предоставляет в ваше распоряжение команду `VACUUM`, которая выполняет функции сборщика мусора, исправляя все ошибки или, по крайней мере, большую их часть. С этой командой тесно связана команда `ANALYZE`, служащая для сборки статистики оптимизатора. Обе команды могут выполняться как единое целое: `VACUUM ANALYZE`. Эти действия также выполняются автоматически, по расписанию, фоновым процессом `autovacuum`, который является частью сервера PostgreSQL.

`VACUUM` производит различные операции по очистке; некоторые из них настолько сложны, что их невозможно объяснить, не углубляясь во внутреннее устройство сервера. Алгоритм `VACUUM` был сильно переработан за прошедшие десять лет, чтобы свести к минимуму

применение блокировки таблиц и повысить эффективность, избегая выполнения лишних действий и используя оптимизацию на уровне процессорного кэша второго уровня для необходимой работы.

Многие опытные администраторы PostgreSQL предпочитают запускать `VACUUM` вручную, несмотря на то что `autovacuum` предусматривает достаточно тонкие настройки и может сэкономить немало времени. Автоматическая сборка мусора в сочетании с ручной обеспечивает контроль и безопасность.

Управление автоматическим обслуживанием базы данных

Автоматическая сборка мусора включена в PostgreSQL 9.0 по умолчанию и в целом хорошо делает свою работу. Однако «в целом» не означает «полностью», так как невозможно предугадать все желания пользователя, например каждый раз выбирать наилучшее время выполнения работ по обслуживанию базы данных.

Подготовка

Чтобы управлять обслуживанием, необходимо определить конечные цели. А для этого понадобится ответить на вопросы:

- Какое время дня лучше всего подходит для тех или иных действий? Когда системные ресурсы бывают более доступны?
- На какие дни недели приходятся пики и спады активного использования базы данных?
- Какие таблицы в большей или в меньшей степени важны для работы приложений?

Как это сделать

Первым делом надо убедиться, что процесс `autovacuum` запущен. В файле `postgresql.conf` должны присутствовать два параметра:

- `autovacuum = on`
- `track_counts = on`

PostgreSQL контролирует `autovacuum` посредством гибкой настройки 35 параметров. Это дает широкие возможности, хотя и не просто в освоении.

В файле `postgresql.conf` можно установить следующие параметры:

- `autovacuum`
- `autovacuum_analyze_scale_factor`
- `autovacuum_analyze_threshold`
- `autovacuum_freeze_max_age`
- `autovacuum_max_workers`
- `autovacuum_naptime`
- `autovacuum_vacuum_cost_delay`
- `autovacuum_vacuum_cost_limit`
- `autovacuum_vacuum_scale_factor`
- `autovacuum_vacuum_threshold`
- `log_autovacuum_min_duration`

Отдельные таблицы управляются «параметрами хранения», которые устанавливаются следующим образом:

```
ALTER TABLE mytable SET (storage_parameter = value);
```

Параметры хранения, имеющие отношение к обслуживанию:

- `autovacuum_enabled`
- `autovacuum_vacuum_cost_delay`
- `autovacuum_vacuum_cost_limit`
- `autovacuum_vacuum_scale_factor`
- `autovacuum_vacuum_threshold`
- `autovacuum_freeze_min_age`
- `autovacuum_freeze_max_age`
- `autovacuum_freeze_table_age`
- `autovacuum_analyze_scale_factor`
- `autovacuum_analyze_threshold`

TOAST-таблицы управляются следующими параметрами:

- `toast.autovacuum_enabled`
- `toast.autovacuum_vacuum_cost_delay`
- `toast.autovacuum_vacuum_cost_limit`
- `toast.autovacuum_vacuum_scale_factor`
- `toast.autovacuum_vacuum_threshold`
- `toast.autovacuum_freeze_min_age`
- `toast.autovacuum_freeze_max_age`
- `toast.autovacuum_freeze_table_age`
- `toast.autovacuum_analyze_scale_factor`
- `toast.autovacuum_analyze_threshold`

Как это работает

Если процесс `autovacuum` запущен, он будет автоматически «просыпаться» каждые `autovacuum_naptime` секунд и определять, запускать ли команды `VACUUM` и/или `ANALYZE`.

Одновременно могут работать не более чем `autovacuum_max_workers` процессов обслуживания. Эти процессы, зависящие от `autovacuum`, выполняя ввод-вывод, накапливают очки стоимости, пока количество очков не достигнет `autovacuum_vacuum_cost_limit`, после

чего процессы «засыпают» на время, равное `autovacuum_vacuum_cost_delay`. Таким образом контролируется потребление ресурсов и предотвращается монополизация процессом `autovacuum` всего дискового ввода-вывода. Увеличивая время `autovacuum_vacuum_cost_delay`, мы замедляем работу `VACUUM`, чтобы не снизить влияние на работу базы данных. Процесс `autovacuum` выполнит команду `ANALYZE`, если сделано хотя бы `autovacuum_analyze_threshold` изменений и обновлено (добавлено, удалено) столько записей относительно общего количества записей в таблице, сколько определяется параметром `autovacuum_analyze_scale_factor`.

Процесс `autovacuum` выполнит команду `VACUUM`, если произведено хотя бы `autovacuum_vacuum_threshold` изменений и обновлено (добавлено, удалено) столько записей относительно общего количества записей в таблице, сколько определяется параметром `autovacuum_vacuum_scale_factor`.

Если задать параметр `log_autovacuum_min_duration`, то каждый раз, когда процесс `autovacuum` будет выполняться дольше установленного значения, это будет отражаться в логе сервера примерно так:

```
2010-04-29 01:33:55 BST (13130) LOG: automatic vacuum of
table
"postgres.public.pgbench_accounts": index scans: 1
  pages: 0 removed, 3279 remain
  tuples: 100000 removed, 100000 remain
  system usage: CPU 0.19s/0.36u sec elapsed 19.01 sec
2010-04-29 01:33:59 BST (13130) LOG: automatic analyze of
table
"postgres.public.pgbench_accounts"
  system usage: CPU 0.06s/0.18u sec elapsed 3.66 sec
```

Большая часть вышеприведенных глобальных параметров может быть также установлена для конкретных таблиц. Допустим, процесс сборки мусора не должен затрагивать таблицу `big_table`. Для этого надо выполнить следующее:

```
ALTER TABLE big_table SET (autovacuum_enabled = off);
```

Можно устанавливать параметры для **TOAST**-таблиц. **TOAST**-таблица содержит значения, выходящие за пределы полей; в документации она также называется таблицей для вспомогательного хранения. Если все значения укладываются в своих полях, **TOAST**-таблица занимает мало места. При очень больших значениях в полях **TOAST**-таблица будет огромной. Алгоритм **TOAST** (*The Oversized-Attribute Storage Technique* – техника хранения значений, выходящих за границы полей) оптимизирован для `UPDATE`. Если в таблице много

обновлений записей, TOAST-таблица не изменяется, поэтому имеет смысл отключить автоматическую сборку мусора в ней таким образом:

```
ALTER TABLE pgbench_accounts
SET ( autovacuum_vacuum_cost_delay = 20
    ,toast.autovacuum_enabled = off);
```

Обратите внимание на то, что автоматическая сборка мусора в TOAST-таблице абсолютно обособлена от сборки мусора в основной таблице, однако при запуске `VACUUM` нельзя включить в процесс или исключить из него конкретную TOAST-таблицу.

Следующий запрос получает на выходе `reloptions` для таблиц и соответствующих TOAST-таблиц:

```
SELECT n.nspname, c.relname,
       pg_catalog.array_to_string(c.reloptions || array(
           select 'toast.' ||
               x from pg_catalog.unnest(tc.reloptions) x),', ' )
       as relopts
FROM pg_catalog.pg_class c
LEFT JOIN
       pg_catalog.pg_class tc ON (c.reltoastrelid =
tc.oid) JOIN
       pg_namespace n ON c.relnamespace = n.oid
WHERE c.relkind = 'r'
AND nspname NOT IN ('pg_catalog', 'information_schema');
nspname |      relname      | relopts
-----+-----+-----
public  | pgbench_accounts  | fillfactor=100,
                                autovacuum_enabled=on,
                                autovacuum_vacuum_cost_
delay=20
public  | pgbench_tellers   | fillfactor=100
public  | pgbench_branches  | fillfactor=100
public  | pgbench_history    |
public  | text_archive       | toast.autovacuum_enabled=off
```

Во время выполнения команды `VACUUM` можно вставлять, изменять и удалять записи, но невозможны такие действия, как `ALTER TABLE` или `CREATE INDEX`. Процесс `autovacuum` способен определять, когда установленная им блокировка мешает пользователю; если `autovacuum` это обнаружит, то отменит блокировку.

Следует заметить, что `VACUUM` во время своей работы не будет сжимать таблицу, если только у нее в конце не образовалось неиспользуемое пространство. Также `VACUUM` не сжимает таблицу, к которой кто-то обращается. Чтобы гарантированно сжать таблицу, используют команду `VACUUM FULL`. При этом вся таблица надолго блокируется, чего следует по возможности избегать. Команда `VACUUM FULL` буквально переписывает каждую строку таблицы и перестраивает все индексы.

В версии 9.0 этот процесс происходит быстрее, чем в предыдущих, но все же при обработке больших таблиц длится долго.

И еще

В файле `postgresql.conf` можно использовать команды `include` следующим образом:

```
include 'autovacuum.conf'
```

При чтении файла `postgresql.conf` на этом месте будет прочитан указанный аргументом команды файл, и содержащиеся в нем команды будут выполнены, как если бы они находились на месте команды `include` в файле `postgresql.conf`.

Это можно использовать для поддержки нескольких режимов сборки мусора. Допустим, база данных обслуживает веб-страницу с высокой нагрузкой в дневные часы. Создадим отдельно дневной профиль – для менее интенсивной сборки мусора и ночной – для более эффективной. Добавим три строки в файл `postgresql.conf`:

```
autovacuum = on
autovacuum_max_workers = 3
include 'autovacuum.conf'
```

и уберем все остальные параметры сборки мусора.

Создадим один файл с названием `autovacuum.conf.day`, содержащий параметры:

```
autovacuum_analyze_scale_factor = 0.1
autovacuum_analyze_threshold = 50
autovacuum_vacuum_cost_delay = 30
autovacuum_vacuum_cost_limit = -1
autovacuum_vacuum_scale_factor = 0.2
autovacuum_vacuum_threshold = 50
```

и другой файл с названием `autovacuum.conf.night`, содержащий параметры:

```
autovacuum_analyze_scale_factor = 0.05
autovacuum_analyze_threshold = 50
autovacuum_vacuum_cost_delay = 10
autovacuum_vacuum_cost_limit = -1
autovacuum_vacuum_scale_factor = 0.1
autovacuum_vacuum_threshold = 50
```

Чтобы включить ночной профиль, нужно выполнить следующее:

```
$ ln -sf autovacuum.conf.night autovacuum.conf
$ pg_ctl -D datadir reload # сервер пересчитывает свою
конфигурацию
```

(зависит от платформы, на которой работает ваш сервер).

Таким способом можно переключать профили, не изменяя файлы конфигурации. Чтобы определить текущий активный профиль, используйте команду `ls -l autovacuum.conf` для вывода детальной информации о файле `autovacuum.conf`. График работы «день/ночь» приведен для примера; едва ли он подойдет для любого сервера.

И еще

Параметр `autovacuum_freeze_max_age` рассматривается в следующем разделе наряду с более сложными параметрами для отдельных таблиц.

Предотвращение «автозаморозки» и повреждения страниц

Необходимость некоторых аспектов `VACUUM` сложно объяснить, при этом иногда они могут мешать работе. Рассмотрим их и попробуем устранить их негативное влияние.

Подготовка

PostgreSQL выполняет регулярную зачистку устаревших идентификаторов транзакций, которая называется «заморозкой». Она нужна для предотвращения «свертывания» счетчика транзакций, которое будет более подробно рассмотрено в следующем разделе.

Строка таблицы PostgreSQL со временем либо помечается к удалению, и тогда ее нужно удалить, либо становится достаточно старой, и тогда ее нужно «заморозить». Удаление и «заморозка» производятся при сборке мусора.

Зачем это нужно знать? Допустим, загружена таблица из 100 миллионов строк. Все работает правильно. Когда строки устаревают в достаточной степени для того, чтобы их «заморозить», следующая команда `VACUUM` или очередное выполнение процесса `autovacuum` для этой таблицы переписывает каждую строку в целях «заморозки» идентификатора транзакции, интенсивно используя ввод-вывод.

Как это сделать

Очевидным способом устранения проблемы представляется выполнение команды `VACUUM` после сильной нагрузки на базу данных.

Разумеется, это не идеальное решение, и для его реализации может не хватать времени.

Многие администраторы по привычке отключают `autovacuum`, поскольку он имеет свойство запускаться в самое неподходящее время. Другое решение описано выше, в разделе «Управление автоматическим обслуживанием базы данных».

«Заморозка» происходит, когда идентификатор транзакции строки становится старше на `vacuum_freeze_min_age` транзакций, чем текущее значение. Обычно при сборке мусора «заморозка» осуществляется в небольших объемах и незаметна. Выше приведен пример, в котором после больших транзакций остается много строк с одинаковыми идентификаторами транзакций, что может вызвать неудобства при «заморозке».

Алгоритм `VACUUM` спроектирован таким образом, чтобы рассматривались лишь те части таблицы, которым необходима чистка. Когда таблица достигает возраста `vacuum_freeze_table_age`, данная оптимизация уже не применяется, и просматривается вся таблица целиком. При этом обнаруживаются строки, требующие «заморозки». В результате использование ввода-вывода резко возрастает.

Попробуем подобрать параметры так, чтобы предотвратить высокую нагрузку при сборке мусора. Параметр `autovacuum_freeze_max_age` определяет, когда таблица будет сканироваться принудительно запущенным процессом `VACUUM`. Если нельзя избежать «заморозки» строк, попробуем выбрать для нее оптимальный момент. Рекомендуется спланировать запуск `autovacuum`, как описано в предыдущем разделе, либо выбирать удобный момент и выполнять команду `VACUUM` вручную.

Сборка мусора также эффективна для того, чтобы подтвердить отсутствие поврежденных страниц, поэтому полезно время от времени сканировать всю базу данных целиком. В этих целях необходимо выполнить следующие команды применительно к каждой базе данных:

```
SET vacuum_freeze_table_age = 0;  
VACUUM;
```

В предыдущих версиях PostgreSQL результаты сборки мусора по всей базе и по каждой отдельной таблице из базы различались. Теперь эти различия устранены.

Если поврежденных блоков нет, следующее полное сканирование можно произвести через два или три месяца. Если поврежденные блоки начали появляться, то, возможно, следует делать это чаще. Обычно повреждение блоков вызвано аппаратным обеспечением,

хотя они и описываются как программные ошибки. Реже причиной становится ошибка PostgreSQL.

В настоящее время не существует простых способов исправить поврежденные страницы. Есть методы исследования и извлечения части данных, например с помощью инструмента `pageinspect`, разработанного автором.

Предупреждение «свертывания» счетчика транзакций

Для многих слова «свертывание счетчика транзакций» звучат как название болезни внесезонного происхождения. Употреблением этого термина говорящий обычно подчеркивает свою техническую грамотность. Разберемся, что подразумевается под свертыванием счетчика транзакций и как с этим бороться.

Подготовка

Прежде всего, взгляните, случалось ли вам видеть следующее сообщение:

```
WARNING: database "postgres" must be vacuumed within XXX
transactions.
HINT: To avoid a database shutdown, execute a database-wide
VACUUM in that database.
You might also need to commit or roll back old prepared
transactions.
```

Или, что еще хуже, следующее сообщение:

```
ERROR: database is not accepting commands to avoid
wraparound data loss in database "template0"
HINT: Stop the postmaster and use a standalone backend to
vacuum that database.
You might also need to commit or roll back old prepared
transactions.
```

Если нет, то не требуется предпринимать какие-либо действия помимо обычного регулярного обслуживания. Приведенные выше сообщения выводятся для пользователей, а также в лог сервера.

Как это сделать

Если вы пользуетесь услугами технической поддержки, самое время обратиться за помощью. Паниковать не надо, но если легкомысленно

отнестись к ситуации, она может серьезно ухудшиться. Рассмотрим способ исправить положение.

Если получено сообщение «WARNING: ...», воспользуйтесь обоими советами, содержащимися в нем. Во первых, выполните команду `VACUUM` для соответствующей базы данных; замените `postgres` на имя вашей базы.

В противном случае выполните следующее:

```
$ vacuumdb postgres
```

или следующее:

```
psql -c "VACUUM" postgres
```

либо воспользуйтесь инструментом из своего администраторского инвентаря для сборки мусора в базе данных.

Далее следуйте инструкциям, описанным в разделе «Удаление ранее подготовленных транзакций».

Как это работает

PostgreSQL использует 32-разрядные идентификаторы транзакций, поэтому всего может быть лишь 231 (примерно 2 млрд) идентификаторов транзакций. По достижении максимального значения счет начинается сначала. Причина в том, что увеличение разрядности идентификаторов имеет свои недостатки. Но и использование четырехбайтовых идентификаторов – не идеальный вариант.

Сервер PostgreSQL спроектирован с учетом цикличности счетчика идентификаторов транзакций. При правильном обслуживании все будет работать вечно, и внутреннее устройство сервера не должно заботить пользователя. В этих целях необходимо регулярно выполнять сборку мусора.

И еще

Если получено вышеприведенное сообщение «ERROR: ...» и в базе данных не выполняются команды, вы, возможно, захотите узнать, что означают слова `standalone backend` и как с помощью этого выполнить сборку мусора.

«Standalone backend» (автономный сеанс) – это подключение к базе данных только одного пользовательского процесса. В ОС *nix есть эквивалент: `runlevel 1`, или однопользовательский режим. База данных доступна лишь одному пользователю.

Для этого используется следующая команда (обратите внимание на то, что опция `--single` должна быть самой первой):

```
$ postgres --single -D /full/path/to/datadir postgres
```

после чего выводится приглашение командной строки:

```
PostgreSQL stand-alone backend 9.0
backend>
```

и тогда можно выполнить команду `VACUUM` следующим образом:

```
PostgreSQL stand-alone backend 9.0
backend> VACUUM;
backend>
```

Затем нажмите CTRL+D (или ту комбинацию клавиш, которая соответствует EOF в вашем терминале) однократно или дважды, если команда `postgres` выполнялась с опцией `-j`.

Прочитайте следующий раздел, «Удаление ранее подготовленных транзакций», и проверьте, есть ли таковые.

Смотрите также

Также вам может пригодиться раздел «Предупреждение “автозамок” и повреждения страниц».

Удаление ранее подготовленных транзакций

Если вы перешли к этому разделу книги по ссылке из других разделов, то вы можете не знать, что такое подготовленные транзакции, тем более старые.

К счастью, подготовленные транзакции выполняются не просто так, а только в особых случаях. Если вы не знаете, о чем речь, ничего страшного – вам это знать необязательно, и, скорее всего, у вас нет никаких подготовленных транзакций.

Подготовленные транзакции являются частью техники «двух-фазного коммита», известного также как **2PC**. Фиксация транзакции выполняется за две стадии, а не за одну, позволяя синхронизировать фиксации в нескольких базах данных. Типичный случай применения данной техники – объединение нескольких «менеджеров ресурсов» через протокол XA, обычно предоставляемый менеджером транзакций (**Transaction Manager, TM**), который используется в **Java Transaction API (JTA)** и в других библиотеках. Если вы ничего из этого не поняли, то, вероятно, у вас нет подготовленных транзакций.

Подготовка

Сначала проверьте значение параметра `max_prepared_transactions`. Если оно равно 0, подготовленных транзакций нет. Если вы все еще не понимаете, о чем речь, то установите значение равным 0.

```
SHOW max_prepared_transactions;
```

Если значение отлично от 0, проверьте, присутствуют ли подготовленные транзакции. В качестве примера приведем команду запроса, аналогичную той, которую вы можете вывести:

```
postgres=# SELECT * FROM pg_prepared_xacts;
-[ RECORD 1 ]-----
transaction | 121083
gid         | prepl
prepared    | 2010-03-28 15:47:57.637868+01
owner       | postgres
database    | postgres
```

Значение `gid` (глобальный идентификатор) обычно сгенерировано автоматически.

Как это сделать

Удаление подготовленных транзакций также называют разрешением сомнительных подготовленных транзакций. Транзакция не была фиксирована и не была отменена. Работа сервера базы данных или менеджера транзакций завершилась аварийно, и транзакция «застряла» между двумя стадиями процесса двухфазного коммита.

Если в пуле 100 активных соединений и где-то произошла авария, то несколько транзакций могли остаться незавершенными; их количество зависит от среднего времени выполнения транзакции.

Для разрешения транзакции следует определить, нужна ли она. Для этого необходимо проверить, что случилось с сервером.

Если вы хотите зафиксировать транзакцию, выполните следующее:

```
COMMIT PREPARED 'prepl';
```

Если требуется отменить изменения, выполните:

```
ROLLBACK PREPARED 'prepl';
```

Как это работает

Подготовленные транзакции сохраняются в случае аварии, поэтому от них нельзя избавиться, просто перезагрузив сервер. Подготовленная транзакция имеет внутренний идентификатор транзакции и внешний глобальный идентификатор. И тот и другой можно

использовать, чтобы определить, какие ресурсы блокированы и как разрешить транзакции.

И еще

Если вы не уверены, в чем именно состояла транзакция, – проверьте, хотя это займет время. Представление `pg_locks` показывает блокировки, установленные подготовленными транзакциями. Полный отчет о блокировках можно получить при помощи запроса:

```
postgres=# SELECT l.locktype, x.database, l.relation,
l.page,
l.tuple,l.classid, l.objid,
l.objsubid,
l.mode, x.transaction, x.gid,
x.prepared,
x.owner
FROM pg_locks l JOIN pg_prepared_xacts x
ON l.virtualtransaction = '-1/' ||
x.transaction::text;
```

В документации говорится, что `pg_locks` можно объединить с `pg_prepared_xacts`, но умалчивается о том, что при объединении по идентификатору транзакций все, что можно увидеть из результата, – что блокировка установлена, если нет блокировки отдельных строк. Блокировки указаны как установленные виртуальной транзакцией. Более простой запрос:

```
postgres=# SELECT DISTINCT x.database, l.relation
FROM pg_locks l JOIN pg_prepared_xacts x
ON l.virtualtransaction = '-1/' ||
x.transaction::text
WHERE l.locktype != 'transactionid';
 database | relation
-----+-----
 postgres | 16390
 postgres | 16401
(2 rows)
```

Здесь видно, какие отношения в каких базах затронуты оставшимися подготовленными транзакциями. Чтобы определить имена этих баз данных, нужно сначала подключиться к ним.

Затем можно полностью просмотреть каждую из этих баз в поисках изменений:

```
SELECT * FROM table WHERE xmin = 121083;
```

Так мы получим все строки в таблице, вставленные транзакцией 121083, номер которой взят из поля `transaction` таблицы `pg_prepared_xacts`.

Для любителей временных таблиц

Если вы активно используете в своих приложениях временные таблицы, возможно, вам требуется выполнять некоторые дополнительные действия применительно к ним.

Как это сделать

Необходимо проверить следующее:

1. Производится ли сборка мусора (ручная – с помощью `VACUUM` или автоматическая – с помощью `autovacuum`).
2. Количество и размер активных временных файлов.
3. Параметры использования оперативной памяти. Подумайте об увеличении параметра `temp_buffers` в разумных пределах.
4. Отдельные устройства для временных файлов. Если загрузка от запросов в системе велика, то обмен ввода-вывода временными данными может превысить по объему ввод-вывод постоянными данными и индексами. В таком случае понадобится создавать новые табличные пространства на отдельных дисках. Убедитесь, что настройка параметра `temp_tablespaces` разрешает использование дополнительных табличных пространств.

Как это работает

В версии 9.0 сервера PostgreSQL при создании временной таблицы добавляются строки в таблицы каталогов `pg_class` и `pg_attribute`. В конце концов эти таблицы каталогов и их индексы начинают раздуваться (данная проблема будет рассмотрена в следующих разделах). Чтобы контролировать этот рост, нужно выполнять ручную (`VACUUM`) или автоматическую (`set autovacuum = on` в файле `postgresql.conf`) сборку мусора. Пользователь не может выполнять команду `ALTER TABLE` для системных таблиц, поэтому изменять настройки `autovacuum` для отдельных таблиц нельзя.

При запуске `VACUUM` для системных таблиц убедитесь, что ни одна таблица не пропущена. Полный список нужных таблиц можно получить следующим запросом:

```
postgres=# SELECT relname, pg_relation_size(oid)
           FROM pg_class
           WHERE relkind in ('i','r') and
          relnamespace = 11
```

ORDER BY 2 DESC;	
relname	pg_relation_size
pg_proc	450560
pg_depend	344064
pg_attribute	286720
pg_depend_depender_index	204800
pg_depend_reference_index	204800
pg_proc_proname_args_nsp_index	180224
pg_description	172032
pg_attribute_relid_attnam_index	114688
pg_operator	106496
pg_statistic	106496
pg_description_o_c_o_index	98304
pg_attribute_relid_attnum_index	81920
pg_proc_oid_index	73728
pg_rewrite	73728
pg_class	57344
pg_type	57344
pg_class_relname_nsp_index	40960

...(часть вывода)

Эти значения указаны для только что созданной базы данных. Таблицы могут сильно разрастись без надлежащего обслуживания. Известен случай, когда один индекс вырос до 11 ГБ.

Обнаружение и исправление раздутых таблиц и индексов

Сервер PostgreSQL использует мультиверсионную систему управления параллельным выполнением операций (**MVCC, Multi-Version Concurrency Control**), позволяющую пользователям считывать строки в то время, как другие пользователи их изменяют. Это важное свойство сервера для обеспечения согласованности приложений, поскольку оно позволяет достичь:

- повышения производительности за счет снижения количества блокировок;
- резкого снижения вероятности взаимных блокировок;
- упрощения разработки приложений и управления ими.

Система MVCC является одной из основных составляющих PostgreSQL и не может быть выключена, тем более что не в ваших интересах ее отключать. Полного понимания внутреннего устройства MVCC от администратора базы данных не требуется. Однако удобства, создаваемые системой, имеют и обратную сторону: при

выполнении команды `SQL UPDATE` таблицы могут увеличиваться в размере, поскольку содержат устаревшие версии строк. После выполнения команды `DELETE` и отмены команды `INSERT` занимаемое пространство не уменьшается до следующей сборки мусора. Для освобождения пространства используется `VACUUM`, а также свойство внутреннего механизма, называемое `HOT`, которое производит заметную часть работы автоматически.

В дальнейшем вы познакомитесь с инструментами дефрагментации, сжатия, реорганизации и оптимизации таблиц. Все это необходимо, но не следует слишком тревожиться из-за того, что базы данных PostgreSQL требуют сборки мусора. Некоторые пользователи тратят массу времени на зачистку старых версий строк, уделяя этому повышенное внимание и в то же время жалуясь, что это отнимает много ресурсов. Вряд ли такое поведение можно считать разумным.

В этом разделе речь пойдет об оценке раздувания таблиц и индексов и о том, когда необходимо выполнять сборку мусора.

Как это сделать

Наилучший способ оценить ситуацию – посмотреть на нее с точки зрения процесса `autovacuum`. Следующий запрос приведен Грегом Смитом (Greg Smith) в его книге «PostgreSQL 9.0 High Performance», выпущенной издательством «Packt». Значения выведены из документации `autovacuum`.

```
CREATE OR REPLACE VIEW av_needed AS
SELECT *,
       n_dead_tup > av_threshold AS "av_needed",
       CASE WHEN reltuples > 0
            THEN round(100.0 * n_dead_tup / (reltuples))
            ELSE 0
       END AS pct_dead
FROM
(SELECT
  N.nspname, C.relname,
  pg_stat_get_tuples_inserted(C.oid) AS n_tup_ins,
  pg_stat_get_tuples_updated(C.oid) AS n_tup_upd,
  pg_stat_get_tuples_deleted(C.oid) AS n_tup_del,
  pg_stat_get_tuples_hot_updated(C.oid)::real /
  pg_stat_get_tuples_updated(C.oid) AS HOT_update_ratio,
  pg_stat_get_live_tuples(C.oid) AS n_live_tup,
  pg_stat_get_dead_tuples(C.oid) AS n_dead_tup,
  C.reltuples AS reltuples, round(
    current_setting('autovacuum_vacuum_
threshold')::integer
```

```

+current_setting('autovacuum_vacuum_scale_
factor')::numeric
* C.reltuples)
  AS av_threshold, date_trunc('minute', greatest(pg_
stat_get_last_vacuum_time(C.oid,pg_stat_get_last_
autovacuum_time(C.oid))) AS last_vacuum, date_
trunc('minute',greatest(pg_stat_get_last_analyze_time(C.
oid),pg_stat_get_last_analyze_time(C.oid))) AS last_analyze
FROM pg_class C LEFT JOIN
pg_index I ON C.oid = I.indrelid
LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
WHERE C.relkind IN ('r', 't')
AND N.nspname NOT IN ('pg_catalog', 'information_
schema') AND
N.nspname !~ '^pg_toast'
) AS av
ORDER BY av_needed DESC,n_dead_tup DESC;

```

Полученное представление можно использовать для того, чтобы просматривать отдельные таблицы следующим образом:

```

postgres=# \x
postgres=# SELECT * FROM av_needed
                WHERE relation = 'public.pgbench_
accounts';
-[ RECORD 1 ]-----+-----
nspname       | public
relname       | pgbench_accounts
n_tup_ins     | 100001
n_tup_upd     | 117201
n_tup_del     | 1
hot_update_ratio | 0.123454578032611
n_live_tup    | 100000
n_dead_tup    | 0
reltuples     | 100000
av_threshold   | 20050
last_vacuum    | 2010-04-29 01:33:00+01
last_analyze   | 2010-04-28 15:21:00+01
av_needed     | f
pct_dead      | 0

```

Как это работает

Можно сравнить количество устаревших версий строк (значение `n_dead_tup`) с пороговым значением, `av_threshold`.

При выполнении вышеприведенного запроса не принимаются во внимание пороговые значения, установленные для отдельных таблиц. При желании это можно исправить, но главная функция запроса – помочь оценить ситуацию принципиально.

В полученной таблице приведены данные для операций вставки, обновления и удаления записей, – это поможет лучше оценить нагрузку. Значение `HOT_update_ratio` показывает долю изменений строк с использованием техники HOT, которая позволяет таблице очищаться автоматически в ходе изменений. Чем ближе это значение к единице, тем меньше необходимости в ручной сборке мусора. Если же значение ближе к нулю, то `VACUUM` или `autovacuum` должны выполняться чаще. Значение `HOT_update_ratio` никогда не достигает 1.0; если оно больше 0.95, это уже превосходный результат.

Изменения с использованием HOT происходят, когда команда `UPDATE` изменяет значения в никем не индексируемых полях. Если изменяется значение в индексируемом поле, HOT не используется, и задействуется больше ресурсов. Поэтому правильное построение индексов может увеличить производительность при изменениях и снизить потребность в обслуживании. Если HOT используется, но, по вашему мнению, недостаточно, то можно попробовать уменьшить значение параметра `fillfactor` для таблицы. Помните, что это касается наиболее активно используемых таблиц. Таблицы, к которым редко обращаются, не требуют тонкой настройки.

Итак, резюме: при операциях обновления без использования HOT индексы разрастаются. Следующий запрос поможет выяснить размер индекса и как он меняется со временем. Запрос выполняется довольно быстро и может быть использован для того, чтобы определить, меняются ли размеры индексов со временем.

```
SELECT
    nspname, relname,
    round(100 * pg_relation_size(indexrelid) /
          pg_relation_size(indrelid)) / 100
      AS index_ratio,
    pg_size_pretty(pg_relation_size(indexrelid))
      AS index_size,
    pg_size_pretty(pg_relation_size(indrelid))
      AS table_size
FROM pg_index I
LEFT JOIN pg_class C ON (C.oid = I.indexrelid)
LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
WHERE
    nspname NOT IN ('pg_catalog', 'information_schema', 'pg_
toast') AND
    C.relkind='i' AND
    pg_relation_size(indrelid) > 0;
```

Еще один способ – использовать модуль `contrib/pgstattuple`, поставляемый с сервером PostgreSQL. Этот модуль предоставляет

разнообразную статистику о происходящем с таблицами и индексами, получая ее в результате просмотра таблицы или индекса целиком и подсчитывая буквально все. Модуль очень полезен, не следует от него отказываться. Нужно только уметь оценивать его полезность: если есть возможность полностью просканировать всю таблицу, то, скорее всего, можно взамен выполнить полную сборку мусора в ней.

При сканировании таблицы используется `pgstattuple()`, как в следующем запросе:

```
test=> SELECT * FROM pgstattuple('pg_catalog.pg_proc');
-[ RECORD 1 ]-----+-----
table_len          | 458752
tuple_count        | 1470
tuple_len          | 438896
tuple_percent      | 95.67
dead_tuple_count   | 11
dead_tuple_len     | 3157
dead_tuple_percent | 0.69
free_space         | 8932
free_percent       | 1.95
```

При сканировании индексов используется `pgstatindex()`:

```
postgres=> SELECT * FROM pgstatindex('pg_cast_oid_index');
-[ RECORD 1 ]-----+-----
version           | 2
tree_level        | 0
index_size        | 8192
root_block_no     | 1
internal_pages    | 0
leaf_pages        | 1
empty_pages       | 0
deleted_pages     | 0
avg_leaf_density  | 50.27
leaf_fragmentation | 0
```

И еще

Возможно, вам будет небезынтересно такое дополнение для Nagios.

Обратите внимание на `check_postgres_bloat`, дополнение библиотеки `check_postgres`. Этот инструмент предоставляет массу возможностей разобраться с раздуванием таблиц и индексов. К сожалению, документация не особенно хороша, но после прочтения этой книги вы сможете понять, о чем речь.

Единственный достоверный способ узнать, что таблица или индекс раздуты, – произвести полное сканирование. Все другие способы являются оценочными.

Обслуживание индексов

Индексы могут создавать проблемы во многих приложениях, активно добавляющих и удаляющих записи в базе данных. Как и таблицы, индексы имеют свойство раздуваться.

В предыдущем разделе говорилось о том, что изменения записей без использования NOT могут вызывать раздувание индексов. Индексы не по первичным ключам раздуваются также при обычных добавлениях записей – это присуще большинству реляционных баз данных.

При автоматической сборке мусора раздутые индексы не обнаруживаются и никакие индексы не перестраиваются. Поэтому необходимо найти способ обслуживания индексов.

Подготовка

PostgreSQL поддерживает перестраивание индексов. Инструмент для клиента базы данных `reindexdb` позволяет выполнять команду `REINDEX` из операционной системы:

```
$ reindexdb
```

При этом выполняется команда SQL `REINDEX` для каждой таблицы в заданной по умолчанию базе данных. Для перестраивания всех индексов во всех базах данных используется опция `-a`:

```
$ reindexdb -a
```

Это отражено в документации. По собственному опыту могу сказать, что большинство индексов не требуется перестраивать, и, кроме того, команда `REINDEX` полностью блокирует таблицу на время своей работы. База данных может оказаться заблокированной на несколько часов, так что советую испробовать другие средства достижения ваших целей. Например, из этого раздела.

Сначала создадим тестовую таблицу, один индекс по первичному ключу и один дополнительный индекс следующим образом:

```
DROP TABLE IF EXISTS test;  
CREATE TABLE test  
(id INTEGER PRIMARY KEY  
, category TEXT  
, value TEXT);  
CREATE INDEX ON test (category);
```

Теперь посмотрим на внутренний идентификатор таблиц, `oid`, и номер текущего файла, или `relfilenode`, следующим образом:

```
SELECT oid, relname, relfilenode
```

```
FROM pg_class
WHERE oid in (SELECT indexrelid
              FROM pg_index
              WHERE indrelid = 'test'::regclass);
```

oid	relname	relfilenode
16639	test_pkey	16639
16641	test_category_idx	16641

(2 rows)

Как это сделать

PostgreSQL поддерживает команду `CREATE INDEX CONCURRENTLY`, которая создает индекс без полной блокировки таблицы. Также PostgreSQL позволяет иметь два индекса с разными именами, но с одним и тем же определением. Фокус в том, чтобы создать новый индекс, идентичный старому, удалить старый индекс и присвоить новому имя старого. И вот результат – индекс перестроен, таблица не была заблокирована:

```
CREATE INDEX CONCURRENTLY new_index
ON test (category);
BEGIN;
DROP INDEX test_category_idx;
ALTER INDEX new_index RENAME TO test_category_idx;
COMMIT;
```

Теперь снова проверим внутренние идентификаторы и получим:

```
SELECT oid, relname, relfilenode
FROM pg_class
WHERE oid in (SELECT indexrelid
              FROM pg_index
              WHERE indrelid = 'test'::regclass);
```

oid	relname	relfilenode
16639	test_pkey	16639
16642	test_category_idx	16642

(2 rows)

Мы видим, что `test_category_idx` – теперь совершенно новый индекс.

Все это замечательно, но не работает для первичных ключей. Почему? Потому что команда `CREATE INDEX CONCURRENTLY` не может использоваться с первичными индексами – по крайней мере, в версии PostgreSQL 9.0¹.

А вот другой трюк, чуть посложнее. Сначала создадим еще один индекс с определением, как у первичного:

```
CREATE UNIQUE INDEX new_pkey ON test (id);
```

¹ Начиная с версии 9.1 допускается использование команды `CREATE UNIQUE INDEX CONCURRENTLY` для перестроивания первичных ключей. – Прим. ред.

и снова проверим внутренние идентификаторы:

```
SELECT oid, relname, relfilenode
FROM pg_class
WHERE oid in (SELECT indexrelid
               FROM pg_index
               WHERE indrelid = 'test'::regclass);
```

oid	relname	relfilenode
16639	test_pkey	16639
16642	test_category_idx	16642
16643	new_pkey	16643

(3 rows)

Теперь переставим два индекса, так что все ограничения для первичного ключа останутся активными, как и внешние ключи, зависящие от него. Поменяем местами значения `relfilenode` следующим образом:

```
BEGIN;
LOCK TABLE test;
UPDATE pg_class SET relfilenode = 16643 WHERE oid = 16639;
UPDATE pg_class SET relfilenode = 16639 WHERE oid = 16643;
DROP INDEX new_pkey;
COMMIT;
```

и убедимся, что получен искомый результат:

```
SELECT oid, relname, relfilenode
FROM pg_class
WHERE oid in (SELECT indexrelid
               FROM pg_index
               WHERE indrelid = 'test'::regclass);
```

oid	relname	relfilenode
16639	test_pkey	16643
16642	test_category_idx	16642
16643	new_pkey	16639

(3 rows)

Совершенно верно: мы только что изменили значения во внутренних таблицах каталогов PostgreSQL. Стоит совершить здесь какую-либо ошибку, и у нас начнутся большие неприятности. Поэтому прежде чем использовать такие приемы, на всякий случай создайте резервную копию базы данных.

Как это работает

Команда `CREATE INDEX CONCURRENTLY` позволяет добавлять, изменять и удалять записи во время создания индекса. Она не может выполняться в другой транзакции, и в любой момент допускается создание только одного индекса.

Поменять местами два индекса легко – тут никаких хитростей не нужно.

Для перестановки местами двух первичных индексов используются сведения о внутреннем устройстве сервера. Индексы не хранят своих номеров, потому их можно обменять без проблем, если они имеют идентичные определения. Если определения индексов различаются и оба индекса принадлежат одному табличному пространству, оно будет испорчено.

И еще

Если вы незнакомы с другими системами управления базами данных, вы можете подумать, что перестраивание индексов во избежание потери производительности необходимо только базам данных PostgreSQL. Однако оно требуется и в других СУБД, хотя об этом не всегда говорится в документации.

Индексы создаются для повышения производительности, и в любой базе данных удаление указателя из индекса вызывает увеличение потребления ресурсов и снижение производительности. PostgreSQL не удаляет из индекса указатели на удаленные строки; таким образом, индекс может заполниться устаревшими указателями. По заполнении блока производится попытка удаления устаревших указателей, но это не мешает им накапливаться в небольших количествах во многих блоках.

Смотрите также

На момент написания этой книги версия 9.0 сервера PostgreSQL готовилась к выпуску. Возможно, в следующих версиях на использование команды `REINDEX CONCURRENTLY` будет наложено меньше ограничений.¹

Обнаружение неиспользуемых индексов

Правильный выбор индексов для повышения производительности – трудная задача. Обычно разработчики и администраторы баз данных находят хороший набор индексов методом проб и ошибок.

¹ Действительно, как уже отмечалось выше, начиная с версии 9.1 стало возможным использование `CREATE UNIQUE INDEX CONCURRENTLY` для перестраивания первичных индексов. – *Прим. ред.*

Существуют инструменты для определения медленных запросов, и часто запрос может стать быстрее с добавлением индекса.

Многие забывают о том, что со временем полезность индексов может меняться. Администратору базы данных необходимо следить за этим и искать оптимальные варианты.

Как это сделать

Сервер PostgreSQL хранит историю обращений к индексу. Чтобы просмотреть эту информацию и определить, не перестал ли индекс быть нужным, воспользуйтесь следующим методом:

```
postgres=# SELECT schemaname, relname, indexrelname, idx_
scan FROM pg_
stat_user_indexes ORDER BY idx_scan;
schemaname |          indexrelname          | idx_scan
-----+-----+-----
public     | pgbench_accounts_bid_idx      |         0
public     | pgbench_branches_pkey         |       14575
public     | pgbench_tellers_pkey          |       15350
public     | pgbench_accounts_pkey         |      114400
(4 rows)
```

Как мы видим, три индекса использовались с разной интенсивностью, а один не использовался совсем. Осталось выяснить, следует ли удалить «ненужный» индекс. Это гораздо более сложный вопрос.

Как это работает

Статистика, собираемая сервером PostgreSQL, содержит различную полезную информацию. Эту статистику администратор может обнулить. Кроме того, эти данные со временем накапливаются, так что создается впечатление, что долго существующие объекты более интенсивно использовались. Если значение `idx_scan` невелико, это может обозначать, что индекс недавно был создан (как, например, в вышеприведенном примере) или, возможно, используется только такими приложениями, которые запускаются лишь в определенные моменты времени (дня, недели, месяца и т. д.).

Также могут использоваться уникальные ограничительные индексы, предотвращающие добавление дублирующих записей в таблицу. Изменение и удаление записей отражается в обновлении `idx_scan`, так как запись должна быть сначала найдена, а добавление записей не отражается. Если записи в таблице только добавляются, индекс будет представляться неиспользуемым.

К некоторым индексам уже давно не было обращений, но они до сих пор выглядят как используемые. Или у таблицы есть другие индексы, которые можно использовать примерно с тем же результатом. Во всем этом следует разобраться, прежде чем действовать.

Смотрите также

Возможно, вы хотите удалить индекс. Вот только бы после этого удалось быстро вернуть его на место, если возникнут проблемы!.. Ведь на воссоздание индекса могут уйти часы, если речь идет о большой таблице. Не стоит унывать – следующий совет поможет избежать нежелательных последствий.

Безопасное удаление ненужных индексов

Безопасное удаление? Имеется в виду, что вы со всеми предосторожностями нажимаете клавишу **Enter** после набора команды `DROP INDEX?..` Отнюдь.

Суть в том, что построение индекса требует много времени, а удаление происходит быстро. Нужен такой способ удаления, чтобы можно было легко отменить эту операцию, если вы произвели ее по ошибке.

Как это сделать

Для начала напомним такую функцию:

```
CREATE OR REPLACE FUNCTION trial_drop_index(iname TEXT)
RETURNS VOID
LANGUAGE SQL AS $$
UPDATE pg_index
SET indisvalid = false
WHERE indexrelid = $1::regclass;
$$;
```

Запустим ее для пробного удаления индекса.

Если после этого производительность снизилась, используем следующую функцию для отмены удаления индекса:

```
CREATE OR REPLACE FUNCTION trial_undrop_index(iname TEXT)
RETURNS VOID
LANGUAGE SQL AS $$
UPDATE pg_index
SET indisvalid = true
WHERE indexrelid = $1::regclass;
$$;
```

Как это работает

Для объяснения нам опять понадобится информация о внутреннем устройстве сервера. Создание индекса с помощью команды `CREATE INDEX CONCURRENTLY` – двухфазный процесс. В первой фазе создается индекс и помечается как недействительный. С этого момента добавления, изменения и удаления записей уже изменяют индекс, а запросы еще не используют его. Должна пройти еще одна фаза, в течение которой таблица сканируется на предмет неиндексированных записей. По завершении второй фазы индекс помечен как действительный и может использоваться запросами.

После того как индекс построен и помечен как действительный, его можно пометить как недействительный – тогда он будет поддерживаться, но не будет участвовать в выполнении запросов. Таким образом, мы легко отключаем индекс и можем так же легко включить его снова, если окажется, что он нам нужен. Благодаря этому удобно проверять, ухудшится ли после удаления индекса производительность наиболее важных запросов.

Планирование обслуживания

Поговорка «не ломается – не трогай» в общем-то справедлива, но только не в отношении обслуживания и профилактики.

Цель обслуживания базы данных состоит в том, чтобы заставить ее работать бесперебойно.

Системы мониторинга не заменяют хорошего планирования. Чем больше может возникнуть внеплановых ситуаций, требующих особого внимания, тем выше вероятность, что несколько непредвиденных событий произойдут одновременно, и в таком случае что-нибудь обязательно сломается. А виноват будет, само собой, администратор. Серьезное отношение к своим профессиональным обязанностям подразумевает грамотное планирование.

Как это сделать

- **Нарушим правило.** Если у вас еще нет резервных копий, сделайте первую *сейчас*. Именно сейчас, не откладывая! Потом вернемся к разговору о планировании обслуживания. Если у вас налажено резервное копирование – замечательно. Вы недолго пробудете в должности администратора баз данных, если потеряете ценную информацию, не имея резервной копии.

В наше время каждая старушка знает, что запасной архив семейных фотоснимков не помешает.

- **Прежде всего, планируйте свое время.** Назначьте дни выполнения определенных действий. Не идите на поводу у собственных систем мониторинга, подскакивая при каждом мелькании индикаторов. Если вас отвлекают другие обязанности, дайте понять, что для бесперебойной работы баз данных нужно регулярное обслуживание.
- **Не бойтесь неожиданностей.** Неизвестное пугает, и впасть в панику легче всего. Вероятно, вашу базу данных не нужно инспектировать ежедневно, но время от времени это стоит делать.

Как это работает

Создайте регулярный график работ по следующим направлениям:

- *Планирование производительности* – сопоставление долгосрочной статистики производительности систем и слежение за ростом объема данных. Прогнозируйте и планируйте поступление новых данных, запуск новых проектов. Можно делать это ежемесячно.
- *Резервные копии, тестовые восстановления, планирование чрезвычайных ситуаций.* Просматривайте планы, тестовые скрипты, очередность резервирования, проверяйте наличие пароля к резервным копиям на внешних системах и т. д. Некоторые системные администраторы выполняют тестовое восстановление каждую ночь, чтобы быть уверенными в том, что восстановление возможно.
- *Сборка мусора и обслуживание индексов* в целях уменьшения раздувания, а также *сбор статистики оптимизации* с помощью команды `ANALYZE`.
- Еще один метод сборки мусора, а именно «заморозку», упомянем особо, чтобы вы о ней не забыли и не жалели об этом впоследствии.
- *Анализ логов сервера.* Сколько раз перезапускался сервер? Точно ли вы знаете все о каждом случае?
- *Безопасность и обнаружение проникновения.* Не взломана ли база данных? Если да, то что именно произошло?
- *Анализ характера использования базы данных.* Если вы толком не знаете, для чего используется база данных, можно поспорить, что вы не сумеете ее правильно настроить и обслуживать.

- *Долгосрочный анализ производительности.* Нередко администратора просят настроить систему, которая медленно работает. Во многих случаях производительность постепенно снижалась в течение длительного периода времени. Не было чрезвычайного события, после которого система замедлила работу, – этот процесс протекал неспешно. Сбор статистики может помочь определить, стала ли система менее «отзывчивой», чем была месяцы или годы назад. Не мешает анализ использования индексов.

Многие из вышеназванных действий упоминаются в этой или других главах книги. Некоторые не рассматриваются, поскольку представляют собой не столько технические задачи, сколько задачи планирования и анализа.

Возможно, у вас найдется время, чтобы обдумать следующее:

- *Качество данных.* Являются ли хранимые данные точными и значимыми? Можно ли улучшить их качество?
- *Значение данных для бизнеса.* Насколько эффективно используются данные для целей предприятия?



ГЛАВА 10.

Производительность и согласованность

Список обсуждаемых вопросов:

- Как найти медленные запросы SQL
- Как получить регулярную статистику из представлений `pg_stat*`
- Как найти причины замедления SQL
- Как уменьшить число возвращаемых строк
- Как упростить сложный SQL
- Как ускорить выполнение запросов, не переписывая их
- Почему запрос не использует индекс
- Как заставить запрос использовать индекс
- Использование оптимистичных блокировок
- Сообщения о проблемах с производительностью

Введение

Производительность и одновременное выполнение запросов – две тесно связанные проблемы: когда улучшается одновременное выполнение запросов, производительность обычно падает, иногда весьма существенно. Если решить проблемы производительности, можно оптимизировать одновременное выполнение запросов.

В этой главе рассматриваются методы обнаружения медленных запросов, а также тех, которые замедляют выполнение других запросов.

Оптимизация производительности не является точной наукой, и вы можете столкнуться с проблемой, которая не будет выявлена ни одним из приведенных способов.

В рецепте «Сообщения о проблемах с производительностью» в конце главы говорится о том, куда обратиться за помощью, если ни один из предыдущих советов не сработал.

Как найти медленные запросы SQL

В базах данных существуют два основных типа замедления.

Первый тип – это единичный запрос, который выполняется настолько медленно, что его использование затруднено. К примеру, информация о клиенте из базы клиентов выдается несколько минут; проверка пароля длится десятки секунд; ежедневный сбор данных – дольше дня. Эти запросы можно выявить, включив на сервере или со стороны клиента логирование запросов, длящихся дольше установленного промежутка времени.

Запросы второго типа выполняются часто (допустим, несколько тысяч раз в секунду) и обычно длятся меньше десяти миллисекунд, но с какого-то момента их выполнение растягивается до десятков и сотен миллисекунд. Такое явление обнаружить гораздо труднее.

Мы рассмотрим несколько способов обнаружения запросов, которые либо замедлились, либо являются причиной замедления работы базы данных.

Подготовка

Подключитесь к базе данных от имени пользователя, запросы которого хотите исследовать, либо как суперпользователь.

Найдите файлы логов PostgreSQL. Обычно они находятся среди других логов, например в системах Linux Debian/Ubuntu – в каталоге `/var/log/postgresql/`.

Включите логирование запросов, длящихся дольше *x* секунд, либо на какое-то время – логирование всех запросов, чтобы получить общую картину.

Как это сделать

Простейший способ выявить отдельные медленные запросы – настройка параметра PostgreSQL для логирования всех медленных запросов. Итак, если вы хотите увидеть все запросы, выполняющиеся дольше 10 с, определите параметр:

```
log_min_duration_statement = 10000;
```

Помните, что значение задается в миллисекундах.

После перезагрузки конфигурации сервера все медленные запросы начнут попадать в лог.

Присвойте параметру `log_duration` значение `on`, чтобы видеть длительность каждого запроса.

Еще один способ найти медленные запросы – поискать их через системное представление `pg_stat_activity`, периодически выполняя запрос:

```
select now()-query_start as running_for, current_query from
pg_stat_activity order by 1 desc limit 5;
```

Этот запрос выбирает из выполняемых в данный момент запросов пять самых «долгониграющих». Вы не сможете узнать таким образом полное время жизни запроса, зато найдете кое-что требующее оптимизации.

Как найти запросы, замедляющие работу сервера

Иногда сам по себе запрос выполняется не слишком медленно, но, запускаясь сотни или тысячи раз, замедляет сервер.

Эти запросы не отразятся в логах, если включить логирование медленных запросов, и все-таки есть способы их обнаружить. Будем повторять запрос:

```
select now()-query_start as running_for, current_query from
pg_stat_activity
```

Если в список часто попадает один и тот же запрос, но значение `running_for` мало, вполне вероятно, что этот запрос использует много ресурсов и по крайней мере частично снижает общую производительность.

Можно собрать такую статистику, выполнив консольную команду:

```
while psql -qt -c "select current_query, now()-query_start
as running_for from pg_stat_activity" >>query_stats.txt ;
do sleep 1; done
```

Каждую секунду будет выполняться запрос и выводить результат в файл `query_stats.txt`.

Через несколько секунд можно остановить выполнение команды нажатием **Ctrl+C** и посмотреть на отсортированный результат:

```
sort query_stats.txt | less
```

в поисках повторяющихся запросов.

- Просмотрите `pg_stat_user_tables` и `pg_statio_user_tables`.
- Другой способ обнаружить такие запросы – искать признаки подозрительного поведения запросов в системных

представлениях `pg_stat*`. Необходимо обратить внимание на следующее:

- в представлении `pg_stat_user_tables` быстрое увеличение значения `seq_tup_read` свидетельствует о том, что происходит много последовательных сканирований. Отношение `seq_tup_read/seq_scan` показывает, сколько кортежей читает каждый `seqscan`;
- в представлении `pg_statio_user_tables` обратите внимание на поля `heap_blks_hit` и `idx_blks_read`, чтобы получить информацию о том, какой объем данных находится в разделяемых буферах PostgreSQL (`heap_blks_hit`) и какой объем будет читаться с диска (`idx_blks_read`). Если много данных читается с диска, то, возможно, следует оптимизировать запросы или увеличить значение параметра `shared_buffers`.

Обнаружив подозрительный запрос, замедлите его, чтобы он попал в логи

Выяснив, что запрос снижает производительность, можно сделать так, чтобы он отображался в логе как медленный. Заблокируйте используемую запросом таблицу на время, чуть большее параметра `log_min_duration_statement`, в файле `postgresql.conf`.

Пример сеанса `psql`, где проверяется таблица `mysuspecttable`:

```
mydb=# begin;
BEGIN
mydb=# lock table mysuspecttable;
LOCK TABLE
select pg_sleep(12);
hannu=# rollback;
ROLLBACK
```

К этому способу можно прибегнуть и в случае, если вы не смогли выделить один подозрительный запрос, но подозреваете, что данная таблица используется неоптимально и желаете найти запросы, которые используют эту таблицу.

Как найти запросы, выполняемые как подготовленные команды

Запрос может быть сначала подготовлен, а затем выполнен, и чтобы найти его, нужно соотнести команду `PREPARE`, используемую для

подготовки запросов, с командой `EXECUTE`, предназначенной для выполнения подготовленных запросов.

Это можно сделать, настроив PostgreSQL для логирования всех запросов и добавив в параметр `log_line_prefix` в конфигурационном файле либо идентификатор процесса ('%p'), либо идентификатор сеанса ('%c'). Тогда, найдя медленно выполняемую команду `EXECUTE`, вы сможете найти соответствующую ей команду `PREPARE`.

Смотрите также

Более подробную информацию об анализе запросов можно найти в конце главы 8.

Как получить регулярную статистику из представлений `pg_stat*`

Ниже будет показано, как регулярным образом собирать статистику, которая позволяет понять, что происходит в базе данных, чтобы оптимизировать медленные запросы.

Подготовка

Найдите пакет `pg_statlogs.tar.gz`.

Создайте каталог для запуска скрипта:

```
mkdir /opt/pg_statlogs
cd /opt/pg_statlogs
tar xzvf pg_statlogs.tgz
```

Создайте схему для сбора снапшотов статистики и генерации из них дельт следующим образом:

```
psql mydatabase <./pg_statlogs_prepare.sql
```

Как это сделать

Наберите в командной строке:

```
psql mydatabase <./pg_statlogs_collect.sql
```

чтобы получить изменения в таблицах `pg_stat*`, произошедшие после предыдущего выполнения команды.

Возможно, следует назначить задание для планировщика, чтобы данные собирались регулярно и можно было наблюдать за тем, что

происходит в разное время суток или в течение недели. Временного разрешения от 5 до 15 минут должно быть достаточно, чтобы на основе собранных данных выяснить, что происходит с вашими таблицами.

Как это работает

Скрипты создают статичные копии таблиц `pg_stat_user_tables`, `pg_statio_user_tables`, `pg_stat_user_indexes` и `pg_statio_user_indexes` при каждом запуске, потом сравнивают с теми, что были получены при предыдущем запуске скриптов, и сохраняют дельты с пометками времени в таблицы `stat_user_tables_delta_log` и `stat_user_indexes_delta_log`, которые в дальнейшем можно анализировать.

Последний по времени набор дельт хранится в таблицах `stat_user_tables_delta` и `stat_user_indexes_delta` и может быть использован внешними системами мониторинга, например Cacti.

И еще

Такие скрипты могут собрать большой объем данных. Это не представляет особой проблемы для производительности сервера, поскольку таблицы логов не имеют индексов и добавление записей происходит быстро, но если места на диске станет недостаточно для хранения всех логов, можно использовать схему ротации, которая будет удалять наиболее старые данные.

Пакет содержит скрипт для этих целей. Выполняйте следующее:

```
psql mydatabase < ./pg_statlogs_rotate.sql
```

каждое воскресенье, и у вас будут храниться только данные за последние четыре недели.

Еще один пакет для сбора статистики

Пакет со схожими функциями для сбора статистики можно найти по адресу: <http://pgstatspack.projects.postgresql.org/>

Как установить причины замедления работы SQL

Команды SQL могут выполняться медленно по разным причинам. Ниже приводится краткий список этих причин и как минимум один способ обнаружения каждой из них.

Обрабатывается слишком много данных

Запускайте запрос с префиксом `EXPLAIN ANALYZE`, чтобы увидеть объем обрабатываемых данных, как в следующем примере:

```
mydb=# EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
QUERY PLAN
-----
Aggregate (cost=4427.27..4427.28 rows=1 width=0) \
    (actual time=32.953..32.954 rows=1
 loops=1)
  -> Seq Scan on t (cost=0.00..4425.01 rows=901 width=0) \
      (actual time=30.350..31.646
 rows=901
 loops=1)
    Total runtime: 33.028 ms
(3 rows)
```

и посмотрите, сколько строк обработано и сколько блоков прочитано, сравнив вывод следующего запроса в системе под нагрузкой и без нагрузки (представления `pg_stat*` – глобальные и собирают данные о всех параллельно выполняемых запросах).

```
select
    s.relid, s.schemaname, s.relname,
    seq_scan, seq_tup_read,
    idx_scan, idx_tup_fetch,
    heap_blks_read, heap_blks_hit,
    idx_blks_read, idx_blks_hit,
    toast_blks_read, toast_blks_hit
from pg_stat_user_tables s
join pg_statio_user_tables sio on s.relid = sio.relid
where s.schemaname = 'public' and s.relname = 't'
```

Например, если вы хотите получить три последних строки из таблицы с миллионом строк, выполните следующий запрос:

```
SELECT * FROM EVENTS ORDER BY ID DESC LIMIT 3;
```

Можно прочитать либо три интересующие вас строки, используя индекс по серийному номеру, либо все строки – и отсортировать по серийному номеру (как в следующем коде), в зависимости от того, можно ли использовать индекс.

```
mydb=# create table events(id serial);
NOTICE: CREATE TABLE will create implicit sequence "events_
id_seq"
for serial column "events.id"
CREATE TABLE
mydb=# insert into events select generate_
series(1,1000000);
INSERT 0 1000000
```

```
mydb=# explain analyse SELECT * FROM EVENTS ORDER BY ID
DESC LIMIT 3;

                                QUERY PLAN
-----
Limit (cost=25500.67..25500.68 rows=3 width=4) \
    (actual time=3143.493..3143.502 rows=3 loops=1)
    -> Sort (cost=25500.67..27853.87 rows=941280 width=4)
        (actual time=3143.488..3143.490 rows=3
        loops=1)
        Sort Key: id
        Sort Method: top-N heapsort Memory: 17kB
        -> Seq Scan on events
            (cost=0.00..13334.80 rows=941280
            width=4)
            (actual time=0.105..1534.418
            rows=1000000
            loops=1)
            Total runtime: 3143.584 ms
(6 rows)

mydb=# create index events_id_ndx on events(id);
CREATE INDEX
mydb=# explain analyse SELECT * FROM EVENTS ORDER BY ID
DESC LIMIT 3;

                                QUERY PLAN
-----
Limit (cost=0.00..0.08 rows=3 width=4) (actual
time=0.295..0.311
rows=3 loops=1)
    -> Index Scan Backward using events_id_ndx on events
    (cost=0.00..27717.34 rows=1000000 width=4) (actual
time=0.289..0.295
rows=3 loops=1)
    Total runtime: 0.364 ms
(3 rows)
```

Разница во времени выполнения запросов 10000-кратная, даже если все данные находятся в оперативной памяти.

Данные не помещаются в оперативную память

Если в разделяемые буферы помещается слишком малая часть данных, одни и те же данные перечитываются с диска снова и снова. Это выражается в том, что значения в одном из полей `heap_blks_read`, `idx_blks_read`, `toast_blks_read` в представлениях `pg_stat*` до и после выполнения запроса существенно различаются.

Если разница между этими значениями велика при первом и втором выполнении запроса, это в порядке вещей, но если и при следующем выполнении запроса значения `*_blks_read` сильно меняются, проблема наличице.

Если вы считаете, что правильно подобрали значение `shared_buffers` и не можете переписать запрос, вам нужен более мощный компьютер.

Запрос возвращает слишком много данных

Иногда ленивые программисты пишут запросы, возвращающие намного больше строк, чем необходимо. Это не слишком заметно, пока объем данных невелик, но когда база расширяется, возникают сложности. Допустим, в базе данных хранятся изображения, а приложение показывает их список. Если за один раз могут быть показаны только десять изображений, то не следует запрашивать более десяти из базы (или одиннадцати, если вы хотите показать ссылку на следующее). Имеет смысл выполнять отдельный запрос `count (*)` для определения общего количества изображений вместо того, чтобы выбирать их все и считать на стороне клиента. Для веб-страниц с высокой нагрузкой предпочтительнее хранить количество изображений в отдельной таблице, чтобы еще больше снизить использование ресурсов для показа графики.

Смотрите также рецепт «Как уменьшить число возвращаемых строк».

Проблемы с блокировками

Использование MVCC позволяет PostgreSQL избежать большей части проблем с блокировками, например таких как недоступность строк для чтения во время записи в них. Однако система вынуждена создавать блокировки при попытке одновременной записи в одну и ту же строку. При записи блокировка длится, пока не завершится транзакция, которая изменяет строку.

Соответственно, если многие запросы могут писать в одну строку, может возникнуть проблема блокировки.

Простейший способ выяснить, многие ли сеансы ждут снятия блокировок, следующий:

```
SELECT * FROM pg_locks WHERE not granted;
```

Если запрос ничего не возвращает или возвращает одну-две строки, то, вероятно, проблемы с блокировками нет.

Чтобы увидеть, какие запросы ждут завершения других запросов, выполните следующее:

```
select
  a1.current_query as blocking_query,
  a2.current_query as waiting_query,
  t.schemaname || '.' || t.relname as locked_table
from pg_stat_activity a1
join pg_locks p1 on a1.procpid = p1.pid and p1.granted
join pg_locks p2 on p1.relation = p2.relation and not
p2.granted
join pg_stat_activity a2 on a2.procpid = p2.pid
join pg_stat_all_tables t on p1.relation = t.relid;
   blocking_query      |      waiting_query      | locked_
table
-----+-----+-----
<IDLE> in transaction | select * from t;        | public.t
<IDLE> in transaction | select count(*) from t; | public.t
(2 rows)
```

Здесь `<IDLE> in transaction` — подключение, в котором открыта консоль, выполнены команды

```
BEGIN;
LOCK t;
```

и ожидается дальнейший ввод команд.

Процессор или устройства ввода-вывода не справляются с нагрузкой

Обычно проблемы связаны с неправильной оптимизацией запросов, но иногда просто не хватает мощности компьютера.

Для быстрой проверки подходит утилита `top`. Наберите в командной строке следующее:

```
user@host:~$ top
```

Прежде всего, обратите внимание на простаивание процессора. Если большую часть времени оно выражается в небольших величинах (скажем, не более 10), то, вероятно, процессор недостаточно мощный.

Если процессор недостаточно занят при высокой нагрузке, то, вероятно, не хватает пропускной способности дисковых устройств. Тогда должно иметь место множество процессов `postgres` со статусом `D`.

Как уменьшить число возвращаемых строк

Помимо того что запрос возвращает слишком много строк, проблема усугубляется тем, что ненужные строки отправляются сервером клиенту, особенно если сервер и клиент находятся на разных компьютерах.

Вот несколько способов уменьшить объем передаваемых данных между сервером и клиентом.

Поиск по тексту возвращает 10 000 документов, но пользователь видит только 20

В таком случае пусть документы сортируются на сервере и запрос возвращает только первые 20:

```
SELECT title, ts_rank_cd(body_tsv, query, 20) AS text_rank
FROM articles, plainto_tsquery('spicy potatoes') AS query
WHERE body_tsv @@ query
ORDER BY rank DESC
LIMIT 20
;
```

Если понадобятся следующие 20, не ограничивайте запрос сорока документами, чтобы выбросить первые двадцать, а используйте `OFFSET 20 LIMIT 20`.

Для большей стабильности – чтобы документы шли в списке в том же порядке, как и в случае, когда используется `OFFSET 20`, – добавьте уникальное поле, например идентификатор документа, к `ORDER BY` в обоих запросах:

```
SELECT title, ts_rank_cd(body_tsv, query, 20) AS text_rank
FROM articles, plainto_tsquery('spicy potatoes') AS query
WHERE body_tsv @@ query
ORDER BY rank DESC, articles.id
OFFSET 20 LIMIT 20
;
```

Приложение запрашивает все строки для сложных расчетов

Постарайтесь выполнить максимум вычислений на сервере.

Вместо того чтобы запросить

```
SELECT * FROM ACCOUNTS WHERE BRANCH_ID = 7;
```

и суммировать строки на стороне клиента, используйте

```
SELECT count(*), sum(balance) FROM ACCOUNTS WHERE BRANCH_ID  
= 7;
```

Изучив немного реализацию языка SQL в сервере PostgreSQL, вы сможете использовать SQL для самых разных вычислений в очень большом объеме.

Если средств SQL окажется недостаточно, прибегните к PL/pgSQL или другому языку из поддерживаемых в PostgreSQL.

Приложение выполняет огромное количество небольших запросов

Это весьма часто происходит при использовании современных ORM (object relational mappers – объектно-реляционных отображений) и других инструментов, которые выполняют за программиста некоторую долю его работы, но при этом скрывают часть происходящего.

Например, вы создаете отчет в HTML с помощью запроса на языке с использованием шаблонов, а потом пишете функцию просмотра шаблона по идентификатору. У вас может получиться форма, которая выполняет запрос для каждой строки, даже когда значения запрашиваются одновременно. Эти запросы могут не представлять проблемы, пока запрашиваемые данные находятся в оперативной памяти, но если запросов множество, возникают осложнения: задержки сети, неоптимальное планирование запросов и т. п.

Здесь возможны два решения:

- заставить ORM кэшировать данные;
- встроить в большой запрос поиск по идентификатору.

Конкретная реализация этих решений зависит от инструмента, но в любом случае оба способа заслуживают изучения, так как с их помощью можно получить значительный выигрыш в скорости и расходовании ресурсов.

Как упростить сложный SQL

В запросах SQL можно встретить два типа осложнений:

- очевидная усложненность, когда запрос состоит из сотен или даже тысяч строк кода. Такой запрос сложно поддерживать, и он может очень медленно выполняться;

- усложненность, скрытая в компонентах: код запроса выглядит простым, но включает представления и функции, которые используют что-то еще. Такой код лучше поддерживается, но может вызывать снижение производительности.

Оба типа запросов могут быть написаны программистами или получены с помощью генераторов запросов.

Подготовка

Сначала убедитесь, что запрос действительно чересчур усложнен.

Запрос, возвращающий много данных, сам по себе не является усложненным. По-настоящему сложный запрос – тот, который соединяет много таблиц сложными способами.

Простейший способ определить, насколько сложен запрос, – использование префикса `EXPLAIN`. Если в выводе большое количество строк, то запрос можно считать сложным.

Как это сделать

Для упрощения запроса обычно следует реструктурировать его так, чтобы выделить части, которые можно определить отдельно и затем использовать в разных частях запроса.

Рассмотрим конкретный пример. Перепишем следующий запрос несколькими способами (полный код содержится в файле `shop_database.tar.gz`). Это запрос типа `pivot` или `crosstab` (строки отображаются в столбцы), объединяющий квартальные прибыли всех магазинов следующим образом:

```
SELECT shop.sp_name AS shop_name,
       q1_nloc_profit.profit as q1_profit,
       q2_nloc_profit.profit as q2_profit,
       q3_nloc_profit.profit as q3_profit,
       q4_nloc_profit.profit as q4_profit,
       year_nloc_profit.profit as year_profit
FROM (SELECT * FROM salespoint ORDER BY sp_name) AS shop
LEFT JOIN (
  SELECT
    spoint_id,
    sum(sale_price) - sum(cost) AS profit,
    count(*) AS nr_of_sales
  FROM sale s
  JOIN item_in_wh iw ON s.item_in_wh_id=iw.id
  JOIN item i ON iw.item_id = i.id
  JOIN salespoint sp ON s.spoint_id = sp.id
  JOIN location sploc ON sp.loc_id = sploc.id
```

```

        JOIN warehouse wh ON iw.whouse_id = wh.id
        JOIN location whloc ON wh.loc_id = whloc.id
    WHERE sale_time >= '2009-01-01'
        AND sale_time < '2009-04-01'
        AND sploc.id != whloc.id
    GROUP BY 1
) AS q1_nloc_profit
ON shop.id = Q1_NLOC_PROFIT.spoint_id
LEFT JOIN (
< similar subquery for 2nd quarter >
) AS q2_nloc_profit
ON shop.id = q2_nloc_profit.spoint_id
LEFT JOIN (
< similar subquery for 3rd quarter >
) AS q3_nloc_profit
ON shop.id = q3_nloc_profit.spoint_id
LEFT JOIN (
< similar subquery for 4th quarter >
) AS q4_nloc_profit
ON shop.id = q4_nloc_profit.spoint_id
LEFT JOIN (
< similar subquery for full year >
) AS year_nloc_profit
ON shop.id = year_nloc_profit.spoint_id
ORDER BY 1
;

```

Выделение части запроса в представление

В предыдущем запросе повторяется почти без изменений та часть, в которой находятся продажи за период, так что имеет смысл выделить эту часть в отдельное представление, а затем использовать его в основном запросе отчета:

```

CREATE VIEW non_local_quarterly_profit AS
SELECT
    spoint_id,
    extract('quarter' from sale_time) as sale_quarter,
    sum(sale_price) - sum(cost) AS profit,
    count(*) AS nr_of_sales
FROM sale s
JOIN item_in_wh iw ON s.item_in_wh_id=iw.id
JOIN item i ON iw.item_id = i.id
JOIN salespoint sp ON s.spoint_id = sp.id
JOIN location sploc ON sp.loc_id = sploc.id
JOIN warehouse wh ON iw.whouse_id = wh.id
JOIN location whloc ON wh.loc_id = whloc.id
WHERE sale_time >= '2009-01-01'
    AND sale_time < '2010-01-01'
    AND sploc.id != whloc.id
GROUP BY 1,2;

```



```

SELECT shop.sp_name AS shop_name,
       q1_nloc_profit.profit as q1_profit,
       q2_nloc_profit.profit as q2_profit,
       q3_nloc_profit.profit as q3_profit,
       q4_nloc_profit.profit as q4_profit,
       year_nloc_profit.profit as year_profit
FROM (SELECT * FROM salespoint ORDER BY sp_name) AS shop
LEFT JOIN non_local_quarterly_profit AS q1_nloc_profit
  ON shop.id = Q1_NLOC_PROFIT.spoint_id
AND q1_nloc_profit.sale_quarter = 1
LEFT JOIN non_local_quarterly_profit AS q2_nloc_profit
  ON shop.id = Q2_NLOC_PROFIT.spoint_id
AND q2_nloc_profit.sale_quarter = 2
LEFT JOIN non_local_quarterly_profit AS q3_nloc_profit
  ON shop.id = Q3_NLOC_PROFIT.spoint_id
AND q3_nloc_profit.sale_quarter = 3
LEFT JOIN non_local_quarterly_profit AS q4_nloc_profit
  ON shop.id = Q4_NLOC_PROFIT.spoint_id
AND q4_nloc_profit.sale_quarter = 4
LEFT JOIN (
  SELECT spoint_id, sum(profit) AS profit
  FROM non_local_quarterly_profit GROUP BY 1
) AS year_nloc_profit
  ON shop.id = year_nloc_profit.spoint_id
ORDER BY 1;

```

Выделение подзапроса в представление не только сократило запрос, но и сделало его проще для понимания и сопровождения.

Использование оператора WITH вместо выделения части запроса в представление

Начиная с версии 8.4 в PostgreSQL можно использовать новый оператор WITH для «встроенного» определения представления, как в следующем примере:

```

WITH nlqp AS (
  SELECT
    spoint_id,
    extract('quarter' from sale_time) as sale_quarter,
    sum(sale_price) - sum(cost) AS profit,
    count(*) AS nr_of_sales
  FROM sale s
  JOIN item_in_wh iw ON s.item_in_wh_id=iw.id
  JOIN item i ON iw.item_id = i.id
  JOIN salespoint sp ON s.spoint_id = sp.id
  JOIN location sploc ON sp.loc_id = sploc.id
  JOIN warehouse wh ON iw.whouse_id = wh.id
  JOIN location whloc ON wh.loc_id = whloc.id
  WHERE sale_time >= '2009-01-01'
)

```

```

        AND sale_time < '2010-01-01'
        AND sploc.id != whloc.id
    GROUP BY 1,2
)
SELECT shop.sp_name AS shop_name,
       q1_nloc_profit.profit as q1_profit,
       q2_nloc_profit.profit as q2_profit,
       q3_nloc_profit.profit as q3_profit,
       q4_nloc_profit.profit as q4_profit,
       year_nloc_profit.profit as year_profit
FROM (SELECT * FROM salespoint ORDER BY sp_name) AS shop
LEFT JOIN nlqp AS q1_nloc_profit
  ON shop.id = Q1_NLOC_PROFIT.spoint_id
  AND q1_nloc_profit.sale_quarter = 1
LEFT JOIN nlqp AS q2_nloc_profit
  ON shop.id = Q2_NLOC_PROFIT.spoint_id
  AND q2_nloc_profit.sale_quarter = 2
LEFT JOIN nlqp AS q3_nloc_profit
  ON shop.id = Q3_NLOC_PROFIT.spoint_id
  AND q3_nloc_profit.sale_quarter = 3
LEFT JOIN nlqp AS q4_nloc_profit
  ON shop.id = Q4_NLOC_PROFIT.spoint_id
  AND q4_nloc_profit.sale_quarter = 4
LEFT JOIN (
    SELECT spoint_id, sum(profit) AS profit
    FROM nlqp GROUP BY 1
  ) AS year_nloc_profit
  ON shop.id = year_nloc_profit.spoint_id
ORDER BY 1
;
```

Использование временных таблиц для частей запроса

Во время оптимизации части запроса могут автоматически материализовываться, но иногда этого не происходит из-за неправильного выбора плана запроса либо из-за недостатка статистики. Или же, как при использовании **GEQO** (**genetic query optimization**, генетической оптимизации запросов), некоторые из возможных планов просто упускают из виду.

Если вы думаете, что материализация (подготовка по отдельности) некоторых частей запроса может быть полезна, можно прибегнуть к помощи временной таблицы, просто выполнив команду `CREATE TEMPORARY TABLE mytemptable01 AS <часть запроса, которую вы хотите материализовать>`, а потом использовать `mytemptable01` в основном запросе вместо материализуемой части. Также допускается создание

индексов для временной таблицы с целью использования в основном запросе.

```
BEGIN;
CREATE TEMPORARY TABLE nlqp_temp ON COMMIT DROP
AS
  SELECT
    spoint_id,
    extract('quarter' from sale_time) as sale_quarter,
    sum(sale_price) - sum(cost) AS profit,
    count(*) AS nr_of_sales
  FROM sale s
  JOIN item_in_wh iw ON s.item_in_wh_id=iw.id
  JOIN item i ON iw.item_id = i.id
  JOIN salespoint sp ON s.spoint_id = sp.id
  JOIN location sploc ON sp.loc_id = sploc.id
  JOIN warehouse wh ON iw.whouse_id = wh.id
  JOIN location whloc ON wh.loc_id = whloc.id
  WHERE sale_time >= '2009-01-01'
    AND sale_time < '2010-01-01'
    AND sploc.id != whloc.id
  GROUP BY 1,2
;
```

Можно создать индексы для таблицы и проанализировать временную таблицу:

```
SELECT shop.sp_name AS shop_name,
       q1_NLP.profit as q1_profit,
       q2_NLP.profit as q2_profit,
       q3_NLP.profit as q3_profit,
       q4_NLP.profit as q4_profit,
       year_NLP.profit as year_profit
FROM (SELECT * FROM salespoint ORDER BY sp_name) AS shop
LEFT JOIN nlqp_temp AS q1_NLP
  ON shop.id = Q1_NLP.spoint_id AND q1_NLP.sale_quarter =
1
LEFT JOIN nlqp_temp AS q2_NLP
  ON shop.id = Q2_NLP.spoint_id AND q2_NLP.sale_quarter =
2
LEFT JOIN nlqp_temp AS q3_NLP
  ON shop.id = Q3_NLP.spoint_id AND q3_NLP.sale_quarter =
3
LEFT JOIN nlqp_temp AS q4_NLP
  ON shop.id = Q4_NLP.spoint_id AND q4_NLP.sale_quarter =
4
LEFT JOIN (
  select spoint_id, sum(profit) AS profit FROM nlqp_
temp GROUP
BY 1
) AS year_NLP
  ON shop.id = year_NLP.spoint_id
```

```
ORDER BY 1
;  
COMMIT; -- здесь заканчивается использование временной  
таблицы
```

Использование материализованных представлений (долгоживущих временных таблиц)

Если часть, заключаемая во временную таблицу, велика, редко изменяется и/или требует сложных вычислений, можно вычислять ее реже, используя технику материализованных представлений. Это представления, которые подготавливаются до использования и либо полностью воссоздаются при изменении используемых данных, либо в некоторых случаях изменяют лишь строки, зависящие от измененных данных. В версии 9.0 PostgreSQL нет явной поддержки материализованных представлений, то есть нельзя написать просто `CREATE MATERIALIZED VIEW AS ...`, но существуют образцы реализации именно такой функциональности. Обсуждение этой темы и конкретные примеры можно найти по адресу: <http://wiki.postgresql.org/wiki/>

Использование функций, возвращающих данные типа «множество» вместо частей запроса

Другой путь получить возможности, подобные тем, что дает применение временных таблиц и/или материализованных представлений, – использовать функции, возвращающие значения типа «множество» (set-returning functions) для некоторых частей запроса.

Например, с помощью функции легко проверить актуальность материализованного представления.

Примеры материализованных запросов и функций, возвращающих данные типа «множество», можно найти в файле `shop_database.tar.gz`.

Как ускорить выполнение запросов, не переписывая их

Если вы не сможете или не захотите переписать запрос, можно все же ускорить его выполнение, используя рассмотренные ниже техники.

Снабжение оптимизатора лучшей информацией

Если при использовании префикса `EXPLAIN ANALYZE` выяснилось, что прогнозы PostgreSQL значительно отличаются от реальной статистики выполнения запросов, нужно настроить сервер для сбора более детальной статистики.

Текущие настройки можно посмотреть, выполнив:

```
show default_statistics_target ;
```

Можно изменить значение либо в файле `postgresql.conf`, либо, если вы хотите изменить его только для одной базы, использовать конструкцию `ALTER DATABASE` следующим образом:

```
ALTER DATABASE mydb SET default_statistics_target = 200 ;
```

Обычно нет смысла делать значение слишком большим для всех таблиц и полей, поскольку это замедлит выполнение команды `ANALYZE`. В PostgreSQL есть более тонкий способ выбрать отдельные поля:

```
ALTER TABLE mytable ALTER col_with_bad_stats SET statistics 500;
```

Новые настройки статистики вступят в силу при следующем выполнении `ANALYZE` в базе данных, поэтому имеет смысл выполнить `ANALYZE` после изменения этих значений.

Если вы меняете параметр `default_statistics_target` базы данных, то изменения вступят в силу после следующего подключения к базе. Поэтому следует либо отключиться и подключиться снова, либо прямо выполнить команду

```
SET default_statistics_target = 300
```

перед запуском `ANALYZE`.

Добавление индекса по нескольким полям специально для запроса

Если запрос выбирает из таблицы `t1` строки по полю `a` и сортирует по полю `b`, то создание следующего индекса позволяет проделать это при просмотре индекса:

```
CREATE INDEX t1_a_b_ndx ON t1(a,b);
```

Добавление специального условного индекса

Если запрос использует некоторое логическое выражение (и особенно если возвращает малое количество строк), то можно

использовать условный индекс по этому выражению, как в следующем примере:

```
CREATE INDEX t1_proc_ndx ON t1(i1)
WHERE needs_processing = TRUE;
```

Это используется в запросе для выбора строк, которые нужно некоторым образом обработать:

```
SELECT id, ... WHERE needs_processing;
```

Кластеризация таблиц по индексам

Обращение к таблице с использованием индекса может быть не очень эффективным, когда индексируемые значения распределены по таблице в случайном порядке. Если вы знаете, что некоторые поля связаны, можно кластеризовать таблицу по индексу, построенному по этим полям. В отношении индекса по нескольким полям можно использовать следующую команду:

```
CLUSTER t1_a_b_ndx ON t1
```

Кластеризация таблицы по индексу переписывает ее в порядке, основанном на данном индексе, так что таблица может быть надолго заблокирована, а потому не делайте этого в нагруженной системе. Кроме того, выполнение команды `CLUSTER` оказывает разовый эффект, и новые строки добавляются не в заданном этой командой порядке. Чтобы сохранить полученное улучшение производительности, кластеризацию нужно время от времени производить заново.

При повторной кластеризации таблицы по индексу нет необходимости указывать в команде имя индекса – можно выполнять ее следующим образом:

```
CLUSTER t1;
```

При этом таблица также полностью переписывается, но если большая ее часть уже упорядочена, выполнение команды может занять меньше времени.

Партиционирование таблиц и изоляция с помощью ограничений целостности

Если запрос выбирает часть огромной таблицы, можно выделить партиции и использовать изоляцию с помощью ограничений целостности (constraint exclusion), чтобы отдельные запросы обращались к партициям.

Партиции в таблицах PostgreSQL версии 9.0 пока не поддерживаются прямо, но существуют базовые возможности для их определения. К сожалению, обсудить этот непростой вопрос в рамках данной книги не представляется возможным. Официальную документацию по использованию партиций в таблицах можно найти по адресу: <http://www.postgresql.org/docs/9.0/interactive/ddl-partitioning.html>

Целая глава другой книги, выпущенной в издательстве Packt, «High Performance PostgreSQL 9.0», посвящена партициям в таблицах. Там тема раскрывается намного глубже, чем в официальной документации.

При частых изменениях используйте параметр *fillfactor* для таблицы

Если вы часто меняете строки в таблице и можете оставлять неизменными индексируемые поля, присвойте параметру *fillfactor* для таблицы значение меньше 100, и тогда сервер сможет менять таблицу с использованием техники HOT, что может на порядок ускорить процесс изменений. Изменения строк с использованием HOT не только не добавляют указатели в индекс, но и быстро убирают мусор на странице для высвобождения места под новые строки.

```
ALTER TABLE t1 SET (fillfactor = 70);
```

Эта команда устанавливает порог заполнения страниц в 70 %, чтобы оставшиеся 30 % могли быть использованы для изменений с использованием HOT.

Более радикальный подход: переписывание схемы

В некоторых случаях имеет смысл переписывание схемы базы данных и сохранение прежнего вида для неизменных запросов с помощью представлений, триггеров, правил и функций.

Например, в ходе рефакторинга базы данных старые запросы должны выполняться, пока происходят изменения. Или, допустим, есть приложение, которое невозможно использовать с текущей схемой, но можно заставить работать, если перераспределить данные между таблицами.

Почему запрос не использует индекс

Ниже будет показано, что делать, если вы считаете, что запрос должен использовать индекс, а этого не происходит.

Тому может быть несколько причин, но самая частая заключается в том, что оптимизатор находит более быстрым и малозатратным план запроса без использования индекса.

Как это сделать

Выполните запрос с принудительным использованием индекса и сравните стоимость выполнения с индексом и без него следующим образом:

```
mydb=# create table itable(id int primary key );
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit
index
"itable_pkey" for table "itable"
CREATE TABLE
mydb=# insert into itable select generate_series(1,10000);
INSERT 0 10000
mydb=# analyse;
ANALYZE
mydb=# explain analyse select count(*) from itable where id
> 500;
```

QUERY PLAN

```
-----
Aggregate (cost=188.75..188.76 rows=1 width=0)
    (actual time=37.958..37.959 rows=1 loops=1)
    -> Seq Scan on itable (cost=0.00..165.00 rows=9500
width=0)
        (actual time=0.290..18.792
rows=9500
loops=1)
```

```
    Filter: (id > 500)
    Total runtime: 38.027 ms
(4 rows)
mydb=# set enable_seqscan to false;
SET
mydb=# explain analyse select count(*) from itable where id
> 500;
```

QUERY PLAN

```
-----
Aggregate (cost=323.25..323.26 rows=1 width=0)
```



```
(actual time=44.467..44.469 rows=1 loops=1)
-> Index Scan using itable_pkey on itable
   (cost=0.00..299.50 rows=9500 width=0)
   (actual time=0.100..23.240 rows=9500 loops=1)
     Index Cond: (id > 500)
Total runtime: 44.556 ms
(4 rows)
```

Как видите, по справедливой оценке оптимизатора запрос лучше выполняется при последовательном чтении.

Как заставить запрос использовать индекс

Здесь пойдет речь о том, как заставить базу данных использовать индекс. На самом деле нельзя принудить PostgreSQL к использованию индекса, но можно «уверить» оптимизатор, что все другие способы намного более затратны.

Подготовка

Сначала убедитесь в том, что использование индекса оправданно. Лучше делать это в системе, используемой для разработки или тестирования, но можно и на рабочем сервере – при соблюдении должной аккуратности. Бывает трудно воспроизвести реальную нагрузку в тестовом окружении; тогда лучше использовать рабочую систему.

Поскольку оптимизатор PostgreSQL не принимает во внимание нагрузку параллельных сеансов, иногда имеет смысл подтасовать статистику, чтобы использовались индексы.

Как это сделать

Выключить `enable_seqscan`

Если вы выполните команду

```
SET enable_seqscan TO false
```

то оптимизатор будет считать все последовательные чтения слишком дорогими. Последовательное чтение выполняется лишь тогда, когда нет других способов:

```
Mydb=# CREATE TABLE table_with_no_index(id int);
CREATE TABLE
mydb=# SET enable_seqscan TO false;
```

```
SET
mydb=# explain select * from table_with_no_index where id
> 10;

               QUERY PLAN
-----
Seq Scan on table_with_no_index (cost=100000000000.00..1000
0000040.00
rows=800 width=4)
  Filter: (id > 10)
(2 rows)
```

Но вполне вероятно, что оптимизатор сочтет более дешевым какой-то другой способ:

```
mydb=# create index table_with_no_index_now_has_one on
table_with_no_index(id);
CREATE INDEX
mydb=# explain select * from table_with_no_index where id
> 10;

               QUERY PLAN
-----
Bitmap Heap Scan on table_with_no_index (cost=10.45..30.45
rows=800
width=4)
  Recheck Cond: (id > 10)
    -> Bitmap Index Scan on table_with_no_index_now_has_one
        (cost=0.00..10.25 rows=800 width=0)
          Index Cond: (id > 10)
(4 rows)
```

Как только вы снова включите `seqscans`, вместо более дорогого (в данном случае) сканирования по индексу по битовой карте будет использоваться последовательное чтение:

```
mydb=# set enable_seqscan to true;
SET
mydb=# explain select * from table_with_no_index where id
> 10;

               QUERY PLAN
-----
Seq Scan on table_with_no_index
(cost=0.00..40.00 rows=800 width=4)
  Filter: (id > 10)
(2 rows)
```

Снизить `random_page_cost`

Для более мягкого уклонения в сторону использования индексов задайте меньшее значение параметра `random_page_cost` — может быть,

даже равное `seq_page_cost`. Это заставит PostgreSQL чаще прибегать к индексам, но не вызовет использования совершенно необоснованных планов, по крайней мере в случаях, когда данные полностью кэшированы сервером или операционной системой.

Стандартные значения этих параметров таковы:

```
random_page_cost = 4;  
seq_page_cost = 1;
```

Попробуйте установить

```
set random_page_cost = 2;
```

и посмотрите, поможет ли это; если нет, присвойте параметру значение 1.

Использование оптимистичных блокировок

Если вы выполняете много транзакций примерно такого рода:

```
BEGIN;  
SELECT * FROM ACCOUNTS WHERE HOLDER_NAME = 'BOB' FOR UPDATE;  
<здесь некоторые вычисления>  
UPDATE ACCOUNTS SET BALANCE = 42.00 WHERE HOLDER_NAME  
= 'BOB';  
COMMIT;
```

то некоторый выигрыш можно получить, если вместо явных блокировок (`SELECT ... FOR UPDATE`) использовать оптимистичные.

При использовании оптимистичных блокировок подразумевается, что никто не будет изменять одну и ту же запись, и на этот предмет проводится проверка во время изменения записи, вместо того чтобы блокировать запись на то время, пока информация обрабатывается на стороне клиента.

Как это сделать

Перепишите приложение так, чтобы предыдущая транзакция выглядела приблизительно следующим образом:

```
BEGIN;  
SELECT A.*, (A.*::text) AS OLDACCINFO  
FROM ACCOUNTS A WHERE HOLDER_NAME = 'BOB';  
<некоторые вычисления>  
UPDATE ACCOUNTS SET BALANCE = 42.00  
WHERE HOLDER_NAME = 'BOB'  
AND (A.*::text) = <OLDACCINFO from select above>;  
COMMIT;
```

Потом убедитесь, что команда `UPDATE` изменила одну строку в коде вашего приложения. Если нет, то запись пользователя `bob` была изменена в промежутке между выполнением `SELECT` и `UPDATE`, и вам, вероятно, придется снова выполнить всю транзакцию.

Как это работает

Строка с данными пользователя `bob` не блокируется на время, пока обрабатываются данные, полученные из запроса; вместо этого предшествующее состояние записи сохраняется в переменной `OLDACCINFO` и впоследствии помогает проверить, не изменилась ли запись.

Можно также сохранить каждое поле (или только те, которые важны) отдельно и затем сравнить с текущими значениями – или проверить автоматически заполняемое поле `last_change`, если оно используется.

И еще

Если вы сумеете передать в базу данных всю нужную для вычислений информацию в виде функции базы данных, то вычисления будут происходить еще быстрее за счет экономии на обмене данными с базой. При работе с PL/pgSQL можно также сохранять план запроса после первого выполнения, чтобы использовать сохраненный план при последующем выполнении, что опять-таки дает выигрыш в производительности.

Итак, приведенная выше транзакция заменяется функцией в базе данных:

```
CREATE OR REPLACE FUNCTION consume_balance (  
    i_username text, i_amount numeric(10,2), max_credit  
    numeric(10,2),  
    OUT success boolean, OUT remaining_balance  
    numeric(10,2)) AS  
$$  
BEGIN  
    UPDATE accounts SET balance = balance - i_amount  
        WHERE username = i_username  
        AND balance - i_amount > - max_credit  
    RETURNING balance  
        INTO remaining_balance;  
    IF NOT FOUND THEN  
        success := FALSE;  
        SELECT balance  
            FROM accounts  
            WHERE username = i_username
```

```
        INTO remaining_balance;  
ELSE  
    success := TRUE;  
END IF;  
END;  
$$ LANGUAGE plpgsql;
```

и вызывается простым выполнением команды

```
SELECT * FROM consume_balance ('bob', 7, 0);
```

на стороне клиента, возвращая значение переменной `success`, которое истинно, если баланс на счете Боба был достаточно велик, и сумму, оставшуюся на балансе Боба после операции.

Сообщения о проблемах с производительностью

Если вам нужно с кем-то посоветоваться по поводу проблем с производительностью, воспользуйтесь списком рассылки по следующему адресу:

<http://archives.postgresql.org/pgsql-performance/>

Не помешает для начала просмотреть архив рассылки на случай, если проблема уже известна.

О том, что обычно следует включать в отчет о проблеме, рассказывается на веб-странице:

http://wiki.postgresql.org/wiki/Guide_to_reporting_problems

Дополнительную информацию, касающуюся производительности, можно найти здесь:

http://wiki.postgresql.org/wiki/Performance_Optimization

Хороший справочник по всем вопросам, связанным с производительностью, – книга «High Performance PostgreSQL 9.0», выпущенная в издательстве Packt.



ГЛАВА 11.

Резервное копирование и восстановление

Рассматриваемые темы:

- Что такое восстановление после сбоя и как его контролировать
- Планирование резервного копирования
- Горячее логическое резервное копирование одной базы данных
- Горячее логическое резервное копирование всех таблиц из табличного пространства
- Резервное копирование определений объектов базы данных
- Автономное горячее физическое резервное копирование базы данных
- Горячее физическое резервное копирование и непрерывное архивирование
- Восстановление всех баз данных
- Восстановление до определенной временной точки
- Восстановление удаленной или поврежденной таблицы
- Восстановление удаленного или поврежденного табличного пространства
- Восстановление удаленной или поврежденной базы данных
- Улучшение производительности резервного копирования и восстановления
- Инкрементальное и дифференциальное резервное копирование и восстановление

Введение

Большинство людей признают важность резервного копирования, но уделяют ему крайне мало времени.

Первый раздел данной главы посвящен восстановлению после сбоя. Нужно разобраться, что происходит при сбое сервера, чтобы понять, что может нуждаться в восстановлении.

Следующий раздел – о планировании. Лучше сначала изучить этот вопрос, нежели спешно создавать резервные копии.

Описываемые механизмы были изначально спроектированы одним из авторов, Саймоном Риггзом (Simon Riggs), для PostgreSQL версии 8.0 в 2004 году, и с тех пор поддерживались по мере роста их популярности (теперь такая поддержка все в большей степени осуществляется благодаря сообществу). Компания 2ndQuadrant также с 2004 года предлагает услуги по восстановлению баз данных, и, к сожалению, уже многим людям это потребовалось по причине отсутствия или поврежденных резервной копии.

Что такое восстановление после сбоя и как его контролировать

Восстановление после сбоя является подсистемой сервера PostgreSQL, которая приходит на выручку, когда сервер аварийно завершает работу или происходит отказ из-за аварии в системе.

Вам будет нелишне узнать, в чем суть данного метода и как им пользоваться.

Как это сделать

Если работа сервера PostgreSQL аварийно завершается, в логе сервера появляется сообщение с уровнем важности `PANIC`. Сервер загружается снова и пытается выполнить восстановление, используя лог транзакций или **Write Ahead Log (WAL, «лог с упреждением»)**.

WAL состоит из последовательности файлов, создаваемых в подкаталоге `pg_xlog` внутри каталога данных PostgreSQL. Каждое изменение базы данных сначала фиксируется в WAL; отсюда и название. Когда транзакция фиксируется, стандартное и безопасное поведение – перенос записей WAL на диск. Если возникнет сбой в работе сервера, произойдет возвращение по записям из WAL к последней зафиксированной транзакции.

Следует отметить, что изменения в базе данных не пишутся на диск в момент фиксации транзакции. Они записываются позже в фоновом процессе.

С какой точки начнется восстановление после сбоя? В логе WAL есть контрольные точки (checkpoints). Длительность процесса восстановления зависит от количества изменений после последней контрольной точки. Именно с контрольной точки можно безопасно начать восстановление, поскольку в этот момент все изменения сохраняются на диске. Однако контрольные точки могут стать узким

местом в работе нагруженной системы из-за интенсивной записи на диск. Есть способы повлиять на это, но следует учитывать, что они оказывают воздействие на систему восстановления.

Два параметра контролируют количество записей WAL между контрольными точками. Параметр `checkpoint_segments` задает количество 16-мегабайтных файлов, которые будут записаны до того, как инициируется создание контрольной точки. Второй параметр, `checkpoint_timeout`, равен времени в секундах до следующей контрольной точки. Контрольная точка создается по достижении одного из этих значений.

Можно увеличить до максимума период создания контрольных точек, задав значения:

```
checkpoint_segments = 1000
checkpoint_timeout = 3600
```

но попробуйте спрогнозировать длительность восстановления при таких настройках и подумайте, нужно ли вам это.

Не забудьте убедиться, что на диске, содержащем каталог `pg_xlog`, достаточно места для `checkpoints_segments * 3 * 16` МБ. Иначе говоря, при `checkpoint_segments = 1000` понадобится *не менее* 32 ГБ места на диске. Если параметр `wal_keep_segments > 0`, то сервер использует до $16 \text{ МБ} * (\text{wal_keep_segments} + \text{checkpoint_segments})$.

Как это работает

Восстановление продолжается до конца лога транзакций. Он пишется непрерывно, поэтому последней будет считаться последняя корректная запись. Каждая запись WAL проверяется на соответствие ее контрольной сумме, поэтому точно известно, корректна ли она. Также каждая запись содержит указатель на предыдущую, поэтому точно известно, составляют ли данные записи непрерывную цепочку. В результате каждое восстановление заканчивается ошибкой чтения записи WAL – это нормальное поведение.

Восстановление может быть очень быстрым, но это зависит от восстанавливаемых операций. Лучший способ проверить скорость восстановления – установка автономного сервера репликаций, описанного в главе 12.

И еще

При воспроизведении лога транзакций могут возникнуть проблемы, и сервер может не запуститься после сбоя.

Иногда в таком случае используют утилиту `pg_resetxlog`, которая удаляет лог транзакций и производит некоторую «уборку» после такого оперативного вмешательства.

Программа `pg_resetxlog` уничтожает изменения в данных, что фактически означает *потерю данных*. Прежде чем использовать этот инструмент, сделайте копию каталога `pg_xlog`. Однако рекомендуется сначала обратиться за помощью, а не прибегать к такой крайней мере. Вы не знаете, поможет ли она решить проблему, но пути назад уже не будет.

Планирование резервного копирования

Этот раздел посвящен прогнозированию и планированию. Если вы прочитаете его перед тем, как начать резервное копирование, – очень хорошо.

Главное, что следует понимать, – планировать нужно не резервное копирование, а восстановление. То, каким образом вы делаете копии, определяет дальнейшие возможности восстановления, поэтому подумайте заранее, чего вы хотите достичь.

Планируя восстановление, нужно учесть, после чего оно может потребоваться.

Рассмотрите следующие основные аспекты:

1. Восстанавливаться будет вся база данных или ее часть?
2. Подлежат восстановлению все данные или только определения объектов?
3. Как распределить контрольные точки?
4. Какова приемлемая скорость восстановления?

Рассмотрите характеристики инструментов, чтобы понять, какие варианты копирования и восстановления доступны. Часто полезно делать резервные копии несколькими способами, чтобы иметь наготове решения для разных ситуаций.

Основные варианты:

- логическое копирование с использованием `pg_dump`;
- физическое копирование файловой системы.

Утилита `pg_dump` представлена двумя вариантами: `pg_dump` и `pg_dumpall`. Для варианта `pg_dump` предусмотрена опция `-F` для записи копий в различных файловых форматах. Выбор файлового формата очень важен, когда дело доходит до восстановления, поэтому подумайте над выбором как следует.

В таблице приводятся доступные функции для разных способов копирования. В следующих разделах эти способы будут описаны детально.

Таблица 11.1.

	Дамп SQL в архивный файл <code>pg_dump -F c</code>	Дамп SQL в скриптовый файл <code>pg_dump -F p</code> или <code>pg_dumpall</code>	Копия файловой системы с помощью <code>pg_start_backup</code>
Тип копирования	Логический	Логический	Физический
Восстановление до временной точки	Нет	Нет	Есть
Копирование всех баз данных	По одной	Да (<code>pg_dump_all</code>)	Да
Копирование всех баз одновременно	Нет	Нет	Да
Выборочное копирование	Да	Да	Нет (см. примечание 3)
Инкрементальное копирование	Нет	Нет	Возможно (см. примечание 4)
Выборочное восстановление	Да	Возможно (см. примечание 1)	Нет (см. примечание 5)
Восстановление удаленной таблицы	Да	Да	Возможно (см. примечание 6)
Восстановление удаленного таблич- ного пространства	Возможно (см. примечание 2)	Возможно (см. примечание 6)	Возможно (см. примечание 6)
Копирование в сжатые файлы	Да	Да	Да
Копирование в несколько файлов	Нет	Нет	Да
Параллельное копирование	Нет	Нет	Да
Параллельное восстановление	Да	Нет	Да
Восстановление на сервере следующей версии	Да	Да	Нет

Таблица 11.1.

Автономное резервное копирование	Да	Да	Да (см. примечание 7)
Определение данных во время копирования	Нет	Нет	Да

Примечания:

1. Если вы сгенерировали скрипт с помощью `pg_dump` или `pg_dumpall` и хотите восстановить один объект, то вам придется углубиться в решение задачи. Понадобится написать скрипт, например на языке Perl, чтобы прочитать файл и выделить нужную часть. Это сложная работа, но она может дать результат быстрее, чем восстановление всей базы на отдельную систему и копирование оттуда нужных объектов.
2. См. раздел «Восстановление удаленного или поврежденного табличного пространства».
3. Выборочное физическое копирование возможно, но приводит к проблемам при восстановлении (см. примечание 6).
4. См. раздел «Инкрементальное и дифференциальное резервное копирование и восстановление».
5. Выборочное восстановление из физической резервной копии невозможно с помощью предоставляемых в данное время утилит.
6. См. раздел «Восстановление удаленного или поврежденного табличного пространства».
7. См. раздел «Автономное горячее физическое резервное копирование базы данных».

Как это сделать

Для создания резервных копий всех баз данных некоторые рекомендуют утилиту `pg_dumpall`. Тем не менее приведем четыре довода против ее использования:

- На выходе получается скрипт, который не может использовать технику параллельного восстановления утилиты `pg_restore`, поэтому восстановление будет производиться медленнее, чем необходимо.
- Утилита `pg_dumpall` создает дампы всех баз данных последовательно. Отсюда закономерный итог:
- Выполнение `pg_dumpall` медленнее, чем выполнение нескольких процессов `pg_dump` параллельно, по одному процессу на одну базу данных.

- Неизвестен конкретный момент времени создания дампа каждой базы данных. Если процесс копирования начался в 4.00 и закончился в 7.00, нельзя сказать с уверенностью, в какой момент появился дамп – известно только, что это было в промежутке между 4.00 и 7.00.
- Опции утилит `pg_dumpall` во многом сходны с опциями утилиты `pg_dump`, но некоторые возможности отсутствуют.

Итак, `pg_dumpall` медленнее копирует, медленнее восстанавливает, предоставляет меньший контроль над процессом создания дампа. Поэтому мы не рекомендуем использовать эту утилиту. Взамен предлагаем один из следующих способов:

- Скопируйте глобальные данные сервера с помощью `pg_dumpall -g`. Потом скопируйте все базы данных параллельно, используя для каждой из них `pg_dump` в отдельном процессе.
- Альтернативный вариант: используйте физическое резервное копирование базы.

Горячее логическое резервное копирование одной базы данных

Логическое резервное копирование данных базы может осуществляться путем создания дампов каждой таблицы.

Как это сделать

Для вышеозначенного используется команда

```
pg_dump -F c > dumpfile
```

или

```
pg_dump -F c -f dumpfile
```

Можно также использовать утилиту `pgAdmin3`, как показано на рис. 11.1:

Как это работает

Утилита `pg_dump` создает один выходной файл. При необходимости можно использовать команду `split(1)` для разделения файла на несколько частей.

Дамп по умолчанию сжат несильно. Можно отключить сжатие или, напротив, сделать более эффективным.



Рис. 11.1

Утилита `pg_dump` выгружает данные из базы с помощью команд SQL. Когда выполняется команда SQL, сервер предоставляет снимок (моментальный снимок) базы данных. Этот снимок пока не может быть предоставлен разным сеансам, поэтому нельзя согласованно копировать из нескольких параллельных процессов `pg_dump` в одной или во многих базах данных.

Момент создания снимка является единственной точкой времени, к которой можно выполнить восстановление. Это момент начала, а не окончания копирования.

Когда выполняется `pg_dump`, таблицы блокируются минимально — чтобы структура таблиц не менялась, пока создаются их дампы. Если изменение структуры таблицы уже началось, процесс `pg_dump` будет приостановлен. Время ожидания можно сократить, используя опцию `--lock-wait-timeout` (время ожидания снятия блокировки).

Утилита `pg_dump` позволяет выборочно копировать таблицы с помощью опции `-t`. Эта же опция позволяет указывать представления и последовательности. Дампы объектов других типов не могут создаваться выборочно с помощью `pg_dump`. Можно использовать

для извлечения фрагментов базы данных функции, доступные на веб-странице по адресу: <http://www.postgresql.org/docs/9.0/static/functions-info.html#FUNCTIONS-INFO-CATALOG-TABLE>

Утилита `pg_dump` работает с базами PostgreSQL ранних версий, поэтому ее можно использовать при переходе от одной версии к другой.

Утилита `pg_dump` не очень хорошо совместима с модулями и не отличается дополнительные таблицы, созданные при установке сторонних пакетов, таких как **PostGIS** или **Slony**, поэтому создает также и дампы этих дополнительных таблиц. В результате не исключены сложности при попытке восстановления, поскольку дополнительные таблицы могли быть созданы в процессе установки программного обеспечения.

И еще

В какое время создана резервная копия с помощью `pg_dump`? Снапшот для дампа делается в момент начала копирования. Время модификации файла показывает момент окончания копирования. Дамп соответствует моменту создания снапшота, и вы захотите знать, когда именно это было. Если вы делаете дампы в виде скрипта, можно добавить опцию `-v` (verbose, «подробно»), и в начале скрипта будет указываться время создания. Дампы произвольного формата также сохраняют время начала копирования, и его можно увидеть следующим образом:

```
pg_restore --schema-only -v dumpfile | head | grep Started
-- Started on 2010-06-03 09:05:46 BST
```

Смотрите также

Заметьте, что `pg_dump` не сохраняет роли (пользователей, группы) и табличные пространства. Эти объекты можно сохранить только с помощью `pg_dumpall`; детальное описание вы найдете в следующих разделах.

Горячее логическое резервное копирование всех баз данных

Если у вас несколько баз данных на сервере PostgreSQL, вы, возможно, захотите создать все их резервные копии одновременно.

Как это сделать

Рекомендуем проделать с каждой из баз данных то же, что вы бы сделали с одной.

Если хотите ускорить процесс, делайте дампы баз данных параллельно.

Затем создайте также дампы глобальных данных:

```
pg_dumpall -g
```

Как это работает

Для создания резервных копий всех баз данных иногда рекомендуют утилиту `pg_dumpall`. Считаем нужным привести четыре довода против ее использования:

- На выходе получается скрипт, который не может использовать технику параллельного восстановления утилиты `pg_restore`, поэтому процесс восстановления будет протекать медленнее, чем необходимо.
- Утилита `pg_dumpall` создает дампы всех баз данных последовательно. В результате:
- Выполнение `pg_dumpall` медленнее, чем выполнение нескольких процессов `pg_dump` параллельно, по одному процессу на одну базу данных.
- Дамп каждой базы данных не сопоставлен конкретному моменту времени. Если процесс копирования начался в 4.00 и закончился в 7.00, нельзя сказать с уверенностью, в какой момент был создан дамп; известно только, что это состоялось в промежутке между 4.00 и 7.00.
- Опции утилит `pg_dumpall` во многом сходны с опциями утилиты `pg_dump`, но некоторые возможности отсутствуют.

Смотрите также

Просмотрите опции в разделе «Горячее физическое резервное копирование и непрерывное архивирование».

Горячее логическое резервное копирование всех таблиц из табличного пространства

Иногда нужно сделать дампы таблиц и данных из табличного пространства. К сожалению, для этого нет простой команды, и придется создать несколько скриптов на подобные случаи.

Как это сделать

Табличное пространство может содержать объекты из нескольких баз данных, поэтому запустите следующий скрипт, чтобы узнать, какие базы данных нужно скопировать:

```
SELECT datname
FROM pg_database
WHERE oid IN (
    SELECT pg_tablespace_databases(ts.oid)
    FROM pg_tablespace ts
    WHERE spcname = 'mytablespacename');
```

Следующая процедура позволяет сделать дампы всех таблиц из табличного пространства, если они принадлежат одной базе данных.

Создайте скрипт `onets.sql`, который выводит список таблиц табличного пространства:

```
SELECT 'pg_dump'
UNION ALL
SELECT '-t ' || spcname || '.' || relname
FROM pg_class t JOIN pg_tablespace ts
ON reltablespace = ts.oid AND spcname = :TSNAME
JOIN pg_namespace n ON n.oid = t.relnamespace
WHERE relkind = 'r'
UNION ALL
SELECT '-F c > dumpfile'; -- dumpfile - это имя выходного файла
```

Выполните следующий запрос, чтобы создать скрипт для `pg_dump`:

```
psql -t -v TSNAME='mytablespace' -f onets.sql > get_my_ts
```

В итоге появятся дампы таблиц табличного пространства, включая определения и данные. Результирующий файл называется `dumpfile` – это определено в последней строке на первом шаге.

```
chmod 755 get_my_ts
./get_my_ts
```

Как это работает

Утилита `pg_dump` позволяет указать в качестве аргументов несколько таблиц. Список таблиц можно сгенерировать и передать из базы данных.

Мы используем именованный параметр в `psql`, чтобы создать скрипт с параметрами, который мы выполним для получения дампа.

Резервное копирование определений объектов базы данных

Иногда полезно получить дампы определений объектов, составляющих базу данных. Благодаря этому становится удобно сравнивать содержимое базы с определениями данных в объектной модели. Также желательно быть уверенными в том, что вы сможете воссоздать объекты, идентичную схему, табличное пространство или базу данных, включая права доступа к объектам.

Как это сделать

Основная команда для копирования только определений:

```
pg_dumpall --schema-only > myscriptdump.sql
```

Определения включают все объекты, роли, табличные пространства, базы данных, схемы, таблицы, индексы, триггеры, уграничения, представления, функции, права доступа и привилегии.

Если вы хотите скопировать определения ролей PostgreSQL, выполните следующее:

```
pg_dumpall --roles-only > myscriptdump.sql
```

Если требуется скопировать определения табличных пространств PostgreSQL:

```
pg_dumpall --tablespaces-only > mytablespaces.sql
```

Если необходимо скопировать и роли, и табличные пространства:

```
pg_dumpall --globals-only > myglobals.sql
```

На выходе получится скрипт, который может прочитать человек и который можно выполнить для воссоздания всех баз данных.

И еще

В PostgreSQL слово «схема» используется также для описания набора связанных друг с другом объектов базы данных, сходного с каталогом и известного также под названием «пространство имен». Убедитесь, что вы правильно понимаете происходящее. Опция `--schema-only` предписывает создать резервную копию «схемы базы данных» – определений всех объектов базы данных (и всех табличных пространств). Для копирования данных и определений только

одного табличного пространства используйте опцию `-s`. Для копирования только определенных одного табличного пространства используйте опции `-s` и `-schema-only`.

Автономное горячее физическое резервное копирование базы данных

Горячее физическое резервное копирование – важная возможность для баз данных.

Физическое копирование позволяет получить точные данные обо всех изменениях в базе. Кроме того, благодаря нему можно сделать резервную копию даже в то время, когда меняются определения объектов базы данных. При этом не устанавливаются блокировки – ограничением являются только ресурсы системы.

Процедура настройки физического резервного копирования несколько сложнее, чем процедура настройки логического копирования. Начнем с простого автономного резервного копирования.

Как это сделать

Следующие шаги предполагают, что заданы переменные среды:

1. Переменная `$PGDATA` содержит путь к каталогу данных PostgreSQL, заканчивающийся символом `/`.
2. Переменная `$BACKUPNAME` содержит имя файла резервной копии.
3. Все необходимые параметры соединений PostgreSQL установлены.

Первое копирование начинается с шага 1. При повторном копировании начинайте с шага 6.

4. Создайте архивный каталог, если таковой отсутствует:

```
cd $PGDATA
mkdir -p ../ standalone
```

5. Задайте переменную `archive_command`. Добавьте в файл `postgresql.conf` следующие строки и перезапустите сервер, если раньше там этих строк не было:

```
archive_mode = on
archive_command = 'test ! ../standalone/archiving_active ||
cp -i %p ../standalone/archive/%f'
```

6. Начните архивирование:

```
mkdir ../standalone/archive  
touch ../standalone/archiving_active
```

7. Начните резервное копирование:

```
psql -c "select pg_start_backup('standalone')"
```

8. Создайте базовую резервную копию – скопируйте файлы данных:

```
tar -cv --exclude=$PGDATA/pg_xlog \  
-f ../standalone/$BACKUPNAME $PGDATA
```

9. Завершите резервное копирование:

```
psql -c "select pg_stop_backup(), current_timestamp"
```

10. Завершите архивирование:

```
rm ../standalone/archiving_active
```

11. Переместите файлы в архивный каталог:

```
mv ../standalone/archive/ archive/
```

12. Добавьте заархивированные файлы к резервной копии:

```
tar -rzf ../standalone/$BACKUPNAME archive/
```

13. Напишите файл конфигурации восстановления `recovery.conf`. Заметьте, что каталог `archive`, упоминаемый здесь, должен соответствовать каталогу, в который копировались файлы на шаге 8.

```
echo "restore_command = 'cp archive/%f %p'" > recovery.conf
```

14. Добавьте файл `recovery.conf` к архиву:

```
tar -rf ../standalone/$BACKUPNAME recovery.conf  
echo "recovery_end_command = 'rm -R archive' " >> recovery.conf
```

15. Сохраните `$BACKUPNAME` где-нибудь в безопасном месте, желательно не на том же сервере.

В результате в отдельном каталоге должен появиться файл `$BACKUPNAME`. Не забудьте выбрать для его хранения надежное место. Файл содержит все необходимое для восстановления, включая параметры восстановления.

Как это работает

Резервная копия, полученная с помощью вышеописанной процедуры, позволяет выполнить восстановление только к одной временной точке. Этой точкой является момент вызова функции `pg_stop_backup()`.

Физическое резервное копирование снимает копию со всех файлов базы данных (см. шаг 8, копирование базы). Этого недостаточно – остальные шаги также необходимы. Простая копия базы данных является несогласованной во времени копией файлов базы данных. Чтобы копия была согласованной во времени, нужно добавить изменения, произошедшие между началом и завершением копирования. Для этого потребуются шаги 7 и 9.

Изменения добавляются в каталог `standalone/archive` в виде архивированных файлов логов транзакций / упреждающего лога (WAL). На шаге 6 создается каталог `archive`. Шаги 1, 2, 3 и 5 нужны для настройки параметров архивирования файлов. При изменении параметра `archive_mode` необходимо перезагрузить сервер, поэтому во избежание перезагрузки приходится прибегать к некоторой хитрости. Эти параметры зависят от существования файла `archiving_active`, присутствие которого включает процесс архивирования, а отсутствие выключает. Параметры можно выбрать другим способом.

На шаге 6 архивирование включается, а на шаге 10 выключается; таким образом, в архив добавляются только те файлы WAL, которые созданы во время резервного копирования. Соответственно, шаги с 1 по 5 – это конфигурирование, а шаги с 6 по 10 – сам процесс резервного копирования. Шаги 11 и последующие – завершающие штрихи, после которых все оказывается в одном файле.

Шаг 11 – архивные файлы перемещаются в каталог с данными, из которого удобнее будет выполнять восстановление. Шаг 12 – в файл с резервной копией добавляются файлы WAL.

Шаги 13, 14 добавляют файл `recovery.conf`, в котором сохранены параметры восстановления, чтобы при использовании резервной копии ничего не надо было делать вручную. За подробным объяснением обратитесь к разделу «Восстановление всех баз данных».

Главное, что следует понимать, – для восстановления необходимы как базовая резервная копия, так и определенный набор файлов WAL. Если что-либо из этого отсутствует, восстановление невозможно. Большая часть шагов нацелена на копирование именно тех файлов WAL, которые вам нужны.

Смотрите также

Используя физическое резервное копирование, обычно прибегают к непрерывному архивированию, потому что оно позволяет выполнить восстановление до любой временной точки.

Горячее физическое резервное копирование и непрерывное архивирование

В этом рецепте описывается резервное копирование с использованием механизма непрерывного архивирования. Назначение непрерывного архивирования состоит в том, чтобы можно было выполнить восстановление до любой временной точки от момента резервного копирования и позже.

Подготовка

Этот рецепт основан на предыдущем, где речь шла об автономном горячем физическом резервном копировании базы данных. Прежде чем читать этот раздел, изучите предыдущий.

Ответьте на несколько вопросов:

- Где будут храниться файлы WAL (будем называть их «архивом»)?
- Как файлы WAL будут добавляться к архиву?
- Где будет храниться базовая резервная копия?
- Как будет создаваться базовая резервная копия?
- Сколько резервных копий будет храниться? Какова политика поддержки архива?

На эти вопросы трудно ответить сразу. Поэтому начнем с практического примера, чтобы посмотреть, как все происходит, а после этого вы выберете подходящий способ.

Как это сделать

Далее мы предполагаем, что на ключевые вопросы были даны следующие ответы:

- Архивом является каталог на удаленном сервере, имя которого `$OTHERNODE`.
- Файлы WAL добавляются к архиву с помощью команды `scp`.
- Базовые резервные копии также хранятся на сервере `$OTHERNODE`.
- Базовые резервные копии создаются с помощью команды `rsync`.
- Будут храниться две резервных копии, одна из которых заменяется при создании новой копии еженедельно по воскресеньям.

Будем считать, что заданы следующие переменные окружения:

- `$PGDATA` содержит путь к каталогу данных PostgreSQL, заканчивающийся символом `/` (обратная косая черта).
- `$OTHERNODE` содержит имя удаленного сервера.
- `$BACKUPNAME` поочередно принимает значения `b1/` и `b2/`, чередующиеся при каждом создании резервной копии. Две резервные копии – абсолютный минимум; возможно, стоит делать больше копий.
- Все параметры подключения PostgreSQL заданы.

Процедура состоит в следующем:

1. Создайте каталоги для резервной копии и архива на удаленном сервере.
2. Задайте параметр `archive_command`. В файл `postgresql.conf` добавьте следующие строки и перезагрузите сервер, если их там не было:

```
archive_mode = on
archive_command = 'scp %p $OTHERNODE:/archive/%f'
```

3. Начните резервное копирование:

```
psql -c 'select pg_start_backup('my backup')'
```

4. Скопируйте файлы с данными (за исключением каталога `pg_xlog`):

```
rsync -cva --inplace --exclude=*pg_xlog* \
${PGDATA}/ $OTHERNODE:$BACKUPNAME/$PGDATA
```

5. Остановите резервное копирование:

```
psql -c 'select pg_stop_backup(), current_timestamp'
```

6. Полезно также добавить файл `README.backup` в каталог с данными перед началом резервного копирования, чтобы он вошел в базовую резервную копию. В файле должны приводиться сведения о том, где находится архив, а также необходимые для его идентификации номера, имена и т. п.

Заметьте, что на этот раз мы не добавляем в резервную копию файл `recovery.conf`, поскольку предполагается, что во время восстановления нам нужна свобода действий, а не одноцелевое решение. Причина в том, что мы не знаем, где, когда и как будет выполняться восстановление, и нам не нужно отвечать на эти вопросы сейчас.

Как это работает

Самое необходимое – иметь для восстановления и базовую резервную копию, и архив. Можете выбирать, где их хранить. Используйте любой способ копирования файловой системы.

Многие системы управления резервным копированием заявлены как имеющие интерфейс или модуль для работы с PostgreSQL, но это чаще всего означает, что они поддерживают логическое копирование. В любом случае не требуется официальная поддержка PostgreSQL со стороны этих систем; не требуется никакой маркировки «Работает с PostgreSQL». Достаточно иметь возможность копировать файлы, чтобы с помощью вышеописанных процессов обеспечивать безопасность базы данных.

Вышеописанная процедура использует простое безопасное копирование файлов, но может также использовать `rsync`. Если пропадет соединение или сервер с резервной копией будет выключен, команда завершится с ошибкой. Когда `archive_command` завершается с ошибкой, она запускается снова, пока не будет успешно выполнена. Сервер не удаляет файлы WAL из каталога `pg_xlog`, пока они не будут успешно архивированы, поэтому каталог может переполниться. Неплохо было бы, если бы команда `archive_command` более оптимально реагировала на такую ситуацию, но предоставим системным администраторам поработать над этим самостоятельно. Типичное решение – настройка оповещений о подобных проблемах, чтобы можно было исправить их вручную. Автоматическое исправление сложно спроектировать правильно, так как трудно воспроизвести ситуацию для тестирования.

С использованием непрерывного архивирования будет генерироваться большое количество файлов WAL. Если значение параметра `archive_timeout` равно 30 с, то за сутки будет создаваться как минимум $2 * 60 * 24 = 2880$ файлов по 16 МБ каждый, что дает в сумме 46 ГБ или больше в день. С приемлемой частотой транзакций сервер может создавать 100 ГБ архивных данных за сутки, и это значение следует использовать как приблизительную оценку, пока оно не будет уточнено в ходе измерений. Частота может быть намного выше – возможны терабайты данных за сутки. Ясно, что мы предпочли бы хранить только те файлы WAL, которые необходимы, поэтому, когда резервная копия уже не будет нужна, понадобится удалить файлы из архива. В каждой базовой резервной копии содержится файл `backup_label`. Имя самого раннего из файлов WAL, соответствующих резервной копии, указывается в первой строке файла `backup_label`. Для

удаления более ранних файлов можно использовать модуль `contrib/pg_archivecleanup`.

При таких размерах архива файлов WAL явно необходимо сжатие файлов. Программы-компрессоры общего назначения для этой цели недостаточно оптимальны. В подобных случаях лучше прибегнуть к специально созданной утилите для сжатия файлов. Программа `pg_lesslog` доступна по адресу: http://pgfoundry.org/frs/?group_id=1000310

Восстановление всех баз данных

Восстановление сервера баз данных полностью – весьма важная операция. В этом разделе будет показано, как ее произвести наиболее простым способом.

Здесь объясняются и некоторые сложные аспекты, но большая их часть освещена в других разделах.

Подготовка

Найдите подходящий сервер, на котором будет выполняться восстановление.

Перед тем как восстанавливать работающий сервер, всегда делайте его резервную копию в текущем состоянии. Не исключено, что проблема, которую вы собираетесь решить с помощью восстановления, только усугубится.

Как это сделать

Логическое (из дампа произвольного формата, созданного командой `dump -F c`):

1. Восстановление всех баз данных означает восстановление каждой отдельной базы данных из дампа. Удостоверьтесь, что вы воспользовались правильной резервной копией:

```
pg_restore --schema-only -v dumpfile | head | grep Started
```

2. Выгрузите глобальные данные:

```
psql -f myglobals.sql
```

3. Выгрузите все базы данных. Для ускорения процесса запускайте восстановление дампов в параллельных процессах. Дампы

можно использовать с удаленного сервера – нет необходимости переносить их между серверами перед использованием. Забудьте, что каждая база данных восстанавливается из отдельного дампа.

```
pg_restore -d postgres -j 4 dumpfile
```

Логическое (из скриптового дампа, созданного командой `dump -F p`): аналогично предыдущему случаю, но на этот раз исполняются скрипты.

1. Убедитесь, что взяли правильную резервную копию. Если вывод следующей команды будет пустым, то в файле нет временной отметки, и тогда придется идентифицировать файл каким-то другим способом.

```
head myscriptdump.sql | grep Started
```

2. Выгрузите глобальные данные из скрипта:

```
psql -f myglobals.sql
```

3. Выгрузите все скрипты:

```
psql -f myscriptdump.sql
```

Логическое (из скриптового дампа, созданного командой `pg_dumpall`). Выполняем следующую процедуру:

1. Убедитесь, что взяли правильную резервную копию. Если вывод следующей команды будет пустым, то в файле нет временной отметки, и тогда придется идентифицировать файл каким-то другим способом.

```
head myscriptdump.sql | grep Started
```

2. Найдите подходящий сервер или создайте новый виртуальный сервер.

3. Выгрузите скрипт

```
psql -f myscriptdump.sql
```

Физическое

Если вы воспользовались рецептом «Автономное горячее физическое резервное копирование базы данных», то восстановление выполняется очень просто:

1. Разверните файл резервной копии в новый каталог.
2. Убедитесь, что взяли правильную резервную копию.

```
$ cat backup_label
```

```
START WAL LOCATION: 0/12000020 (file  
0000000100000000000000012)
```

```
CHECKPOINT LOCATION: 0/12000058
```

START TIME: 2010-06-03 19:53:23 BST

LABEL: standalone

3. Убедитесь, что файлы принадлежат тем пользователям, которым должны принадлежать, что права доступа назначены корректно и ссылки действительны. С этим не должно быть проблем, если вы работаете с базами данных от имени пользователя `postgres`, как рекомендуется.
4. Запустите сервер.

Произвести эту процедуру не составит никакого труда, потому что в рецепте «Автономное горячее физическое резервное копирование базы данных» было сделано все возможное для упрощения использования резервной копии.

Если использовался другой алгоритм, понадобятся дополнительные действия:

1. Отключите все серверы, работающие с каталогом данных.
2. Разверните резервную копию, чтобы все файлы в каталоге данных с соответствующими именами были заменены версиями из резервной копии. (В руководстве предлагается удалить все файлы из каталога, а потом развернуть резервную копию, но это может занять намного больше времени, чем выполнение команды `rsync` без опции `--update`.) Не забывайте, что этот процесс можно распараллелить для ускорения.
3. Убедитесь, что файлы принадлежат тем пользователям, которым должны принадлежать, что права доступа назначены корректно и ссылки действительны. С этим не должно быть проблем, если вы работаете с базами данных от имени пользователя `postgres`, как рекомендуется.
4. Удалите все файлы из каталога `pg_xlog`. (Если резервное копирование выполнялось по алгоритмам, описанным в данной книге, это не понадобится, так как упомянутые файлы не копировались.)
5. Скопируйте самые последние файлы WAL, если таковые есть, с работающего сервера.
6. Добавьте файл `recovery.conf` и назначьте корректные права доступа к нему.
7. Запустите сервер.

Единственное, что нужно обдумать и проверить, – параметры в файле `recovery.conf`. Для наших целей только один из них имеет смысл – `restore_command`.

Параметр `restore_command` отвечает за восстановление файлов WAL. Значение этого параметра – команда, которая должна быть выполнена, когда WAL-файлы извлекаются из архива.

Если вы будете предусмотрительны, то добавите файл `README.backup`, в котором объясняется, какой должна быть команда `restore_command`. В противном случае вам придется запоминать, куда сохраняются WAL-файлы.

Допустим, файлы сохранены в каталоге `/backups/pg/servername/archive`, принадлежащем пользователю `postgres`.

На удаленном сервере `backup1` надо будет добавить в файл `recovery.conf` следующую строку:

```
restore_command = 'scp backup1:/backups/pg/servername/
archive/%f %p'
```

Как это работает

Архитектура PostgreSQL такова, что для восстановления необходим минимум информации. Мы старались как можно меньше загружать пользователя деталями.

- **Логическое восстановление:** команды SQL воссоздают объекты базы данных. Если возникнут проблемы с производительностью, прочитайте раздел о производительности восстановления.
- **Физическое восстановление:** данные изменяются на уровне блоков, так что восстановление может происходить гораздо быстрее логического. Необходимы как базовая резервная копия, так и набор архивных файлов WAL.

В каталоге данных базовой резервной копии находится файл `backup_label`. В нем говорится, что нужно извлечь из архива файл `.backup`, который содержит расположение стартового файла и стоп-адреса WAL для данной базовой резервной копии. Потом происходит восстановление, которое начинается со стартового файла WAL и должно продолжиться до конечного файла.

Позавершении этого процесса файл `recovery.conf` переименовывается в `recovery.done`, чтобы предотвратить повторное восстановление.

Лог сервера регистрирует каждый файл WAL, восстановленный из архива, так что можно наблюдать развитие процесса и скорость восстановления. Можно найти в архиве самый последний файл WAL, чтобы определить, сколько еще будет обработано файлов.

Команда `restore_command` должна вернуть 0, если файл восстановлен успешно, и ненулевое значение, если произошла ошибка. Восстановление происходит до тех пор, пока обнаруживается следующий файл WAL; когда таковой будет обнаружен, в логе появится сообщение об ошибке.

Если какой-то файл WAL утерян или поврежден, восстановление остановится. Последующие изменения не будут применены и, скорее всего, будут утеряны. Самое время звонить в техническую поддержку!

И еще

После начала восстановления можно останавливать и запускать сервер. Это не мешает процессу восстановления.

Во время восстановления можно подключаться к базе данных и выполнять запросы. Это так называемый *режим горячего резервирования* (hot standby mode), описанный в следующей главе.

Смотрите также

Если процесс дошел до стоп-адреса, восстановление можно прервать в любой момент, как описано в следующем разделе.

Восстановление до определенной временной точки

Если в 15.22 в вашей базе данных возникла проблема, а резервное копирование было выполнено в 4.00, вы, вероятно, надеетесь, что есть способ отказаться от изменений, сделанных в промежутке между этими двумя моментами времени. Вам нужно *восстановление до временной точки* – point-in-time recovery (PITR).

К сожалению, если резервное копирование было выполнено с помощью `pg_dump` в 4.00, то вы сможете восстановить только состояние базы на этот момент – и никакой другой. Поэтому термин «PITR» в PostgreSQL синонимичен физическому резервному копированию и восстановлению.

Подготовка

Если резервное копирование было сделано с помощью `pg_dump`, оставьте всякую надежду использовать эту копию как базовую для

восстановления до временной точки! Часто спрашивают, возможно ли это, и ответ всегда отрицательный. А поскольку такой вопрос продолжают задавать снова и снова, мы и настаиваем, чтобы вы заранее планировали резервное копирование.

Во-первых, определите, до какого момента времени вам может понадобиться выполнить восстановление. Если до самого позднего, то вам не требуется PITR – просто восстанавливайте до конца логов.

Как это сделать

Как определить, какая временная точка нас интересует? Момент, на котором восстановление останавливается, называют «целью восстановления» (recovery target). Самый прямой способ основан на временных отметках.

В файле `recovery.conf` нужно добавить (раскомментировать) следующую строку:

```
recovery_target_time = '2010-06-01 16:59:14.27452+01'
```

или подобную. Заметьте, что указанная временная зона должна соответствовать настройкам сервера, на котором велись логи. Она может отличаться от временной зоны сервера, на котором выполняется восстановление.

После этого можно следить за процессом, выполняя запросы в режиме горячего резервирования.

Как это работает

Восстановление выполняется за счет применения записей WAL. Они соответствуют индивидуальным изменениям блоков, поэтому каждой транзакции сопоставлено много записей WAL. Заключительной частью транзакции является запись WAL о фиксации, но может быть и запись об отмене. Каждая запись о завершении транзакции имеет временную отметку, которая позволяет решить, выполнять ли восстановление до этой временной точки.

Для определения нужного момента можно также использовать *идентификатор транзакции* (transaction id, xid), но найти нужный xid довольно сложно, и могут понадобиться внешние записи, если они есть.

Цель восстановления указывается в файле `recovery.conf` и не может быть изменена, пока работает сервер. Если нужно изменить цель восстановления, необходимо остановить сервер, отредактировать

файл `recovery.conf` и запустить сервер. Будьте осторожны, потому что, если вы измените цель восстановления, а восстановление уже миновало этот момент, могут возникнуть ошибки. Если вы определите для параметра `recovery_target_timestamp` значение уже пройденной временной точки, то восстановление будет остановлено почти немедленно, но не на корректной точке остановки. Если вы определите для параметра `recovery_target_xid` значение, которое уже пройдено, то восстановление будет продолжаться до конца логов. Безопасным решением будет отменить восстановление, снова развернуть базовую резервную копию и начать процесс восстановления сначала.

С восстановлением сервера начинается новая последовательность событий. Мы можем вносить новые изменения в базу данных. Эти изменения могут отличаться от тех, которые мы сделали после резервного копирования в предыдущей истории сервера. Таким образом, в разных последовательностях событий мы имеем разные варианты будущего. Мы можем вернуться назад, опять выполнив восстановление и создав новую историю сервера. Это похоже на научную фантастику: есть шанс вернуться в прошлое и предпринять что-то другое. Впрочем, будьте осторожны и не запутайтесь.

И еще

С помощью утилиты `pg_dump` нельзя создать базовую резервную копию для PITR. Причина в том, что лог содержит изменения физических блоков данных, а не логические изменения. Если выгрузить `pg_dump`, то данные могут попасть в другие блоки, и изменения не будут соответствовать данным.

Файлы WAL содержат недостаточно информации для полного восстановления всех команд SQL, произведших изменения. При дальнейшем развитии PostgreSQL необходимая информация может быть добавлена в файлы WAL.

Смотрите также

В версии 9.1 планировалось добавить возможность приостанавливать и возобновлять или прекращать восстановление, а также менять цель восстановления динамически при работающем сервере. Это позволит с большим удобством определять нужную точку завершения восстановления с помощью режима горячего резервирования¹.

¹ См. функции `pg_xlog_replay_pause()`, `pg_xlog_replay_resume()` и `pg_create_restore_point()`, которые появились в версии 9.1. – *Прим. ред.*

Иногда полезно с помощью режима горячего резервирования добиться остановки восстановления. Читайте раздел, описывающий управление режимом горячего резервирования.

Восстановление удаленной или поврежденной таблицы

Предположим, что некая таблица удалена или повреждена. Причиной могли послужить повреждение диска или некорректные удаления и изменения с потерей важных данных.

Обычно проблема решается путем восстановления из резервной копии.

Как это сделать

Есть разные способы действий – выбор зависит от того, какого типа резервные копии есть в наличии. Если существуют резервные копии разных типов, возможны варианты.

Логическое (из резервной копии произвольного формата, созданной с помощью команды `dump -F c`).

Если вы выполнили логическое резервное копирование с помощью утилиты `pg_dump` в файл произвольного формата, можно просто извлечь нужную таблицу из файла дампа следующим образом:

```
pg_restore -t mydroppedtable dumpfile | psql
```

или соединиться с базой данных напрямую, используя опцию `-d`.

Приведенная команда пытается создать заново таблицу и заполнить ее данными. Заметьте, что `pg_restore` с опцией `-t` не извлекает из дампа индексы для таблицы. Это значит, что понадобится немного более сложная процедура, которая зависит от того, восстанавливаем ли мы поврежденную или удаленную таблицу.

Чтобы исправить поврежденную таблицу, надо заместить данные в ней в течение одной транзакции. Специальной опции для этой цели нет, поэтому надо выполнить следующее:

1. Сохранить дамп таблицы таким образом:

```
pg_restore -t mydroppedtable dumpfile > mydroppedtable.sql
```

2. Записать в скрипт `restore_mydroppedtable.sql` следующий код:

```
BEGIN;  
TRUNCATE mydroppedtable;  
\i mydroppedtable.sql  
COMMIT;
```

3. Потом запустить скрипт:

```
psql -f restore_mydroppedtable.sql
```

4. Если таблица была удалена, то нужно:

- Создать новую временную базу данных – назовем ее restorework:

```
CREATE DATABASE restorework;
```

- Восстановить полную схему базы данных:

```
pg_restore --schema-only -d restorework dumpfile
```

5. Создадим дамп нужной таблицы в новом файле. Он будет содержать команды CREATE TABLE, создания индексов, других ограничений и привилегий. Поскольку временная база данных не содержит данных, то опция --schema-only несущественна:

```
pg_dump -t mydroppedtable --schema-only restorework >  
mydroppedtable.sql
```

6. Воссоздадим таблицу в основной базе данных:

```
psql -f mydroppedtable.sql
```

7. Выгрузим только данные таблицы в базу:

```
pg_restore -t mydroppedtable --data-only -d maindb dumpfile
```

Если таблица очень велика, то четвертый шаг может быть проблемным, потому что он воссоздает индексы. Можно разделить скрипт на два: первая часть нужна перед выгрузкой таблицы, а вторая – после. Некоторые усовершенствования для этого приводятся в конце раздела.

Логическое (из скриптового дампа).

Простой способ восстановить одну таблицу из скрипта таков:

1. Найдите подходящий сервер или создайте виртуальный.

2. Выполните скрипт полностью:

```
psql -f myscriptdump.sql
```

3. На восстановленном сервере создайте в новом файле дамп таблицы, ее определений и данных:

```
pg_dump -t mydroppedtable -F c mydatabase > dumpfile
```

4. Теперь воссоздайте таблицу на целевом сервере. Это можно сделать удаленно, не копируя дамп между серверами:

```
pg_restore -d mydatabase -j 2 dumpfile
```

Единственный способ извлечь единственную таблицу из скриптового файла без восстановления всего остального содержимого – написать программу, извлекающую из скрипта только нужные нам части. Этими частями могут быть команды SET где-нибудь в начале файла, данные могут быть в середине, а индексы и ограничения для таблицы – в конце скрипта. Такой способ сложен, в то время как предложенный выше вполне безопасен.

Физическое.

Чтобы восстановить единственную таблицу из физической резервной копии, нам нужно:

1. Найти подходящий сервер или создать виртуальный.
2. Полностью восстановить сервер, включая все базы данных и все таблицы, как было показано в предыдущих разделах, посвященных физическому восстановлению. Если вы хотите остановить восстановление на определенной временной точке, найдите соответствующий раздел в конце этой главы.
3. На восстановленном сервере создайте дампы таблицы, ее определений и данных в новом файле:

```
pg_dump -t mydroppedtable -F с mydatabase > dumpfile
```

4. Теперь воссоздайте таблицу на целевом сервере. Это можно сделать удаленно, не копируя дампы между серверами:

```
pg_restore -d mydatabase -j 2 dumpfile
```

Как это работает

В настоящий момент нет простого способа восстановить одну таблицу одним действием.

Смотрите также

Разделение скрипта `pg_dump` на части «до выгрузки» и «после выгрузки» было предложено одним из авторов данной книги еще для предыдущей версии PostgreSQL – но не было времени доработать это. Можно выполнить аналогичные действия с помощью сторонних инструментов; лучший из виденных автором скриптов для разделения дампа на две части доступен по адресу: http://bucardo.org/wiki/split_postgres_dump

Восстановление удаленного или поврежденного табличного пространства

Иногда требуется полное восстановление одного табличного пространства. Это гораздо проще, чем восстановление одной таблицы.

Как это сделать

Способ зависит от того, какого типа резервная копия у вас есть. Если есть копии разных типов, то возможны варианты.

Логическое (из резервной копии произвольного формата, созданной с помощью команды `dump -F c`).

Если вы выполнили логическое резервное копирование с помощью утилиты `pg_dump` в файл произвольного формата, можно просто извлечь нужные таблицы из файла дампа следующим образом:

```
pg_restore -t mytab1 -t mytab2 - dumpfile | psql
```

или соединиться с базой данных напрямую, используя опцию `-d`.

Разумеется, вы можете не помнить точно, какие таблицы вам нужны. В таком случае понадобится:

1. Найти подходящий сервер или создать виртуальный.
2. Выполнить скрипт полностью, в четыре потока:

```
pg_restore -d mydatabase -j 4 dumpfile
```
3. Когда восстановление завершится, можно создать дамп таблиц, следуя инструкциям, приведенным в разделе «Горячее логическое резервное копирование всех таблиц из табличного пространства».
4. Теперь воссоздайте таблицы на целевом сервере. Это можно сделать удаленно, не копируя дамп между серверами:

```
pg_restore -d mydatabase -j 2 dumpfile
```

Логическое (из скриптового дампа).

Не существует простого способа извлечь нужные таблицы из скриптового дампа. Необходимо использовать следующую процедуру:

1. Найти подходящий сервер или создать виртуальный.
2. Выполнить скрипт полностью:

```
psql -f myscriptdump.sql
```
3. Когда восстановление завершится, можно сделать дамп таблиц, следуя инструкциям раздела «Горячее логическое резервное копирование всех таблиц из табличного пространства».
4. Теперь воссоздайте таблицы на целевом сервере. Это можно сделать удаленно, не копируя дамп между серверами:

```
pg_restore -d mydatabase -j 2 dumpfile
```

Физическое.

Чтобы восстановить одно табличное пространство из физической резервной копии, необходимо:

1. Найти подходящий сервер или создать виртуальный.
2. Полностью восстановить сервер, включая все базы данных и все таблицы, как показано в предыдущих разделах, посвященных физическому восстановлению.

3. Если вы хотите остановить восстановление на определенной временной точке, найдите соответствующий раздел в конце этой главы.
4. На восстановленном сервере создайте дампы таблиц табличного пространства, следуя инструкциям раздела «Горячее логическое резервное копирование всех таблиц из табличного пространства».
5. Теперь воссоздайте таблицы на целевом сервере. Это можно сделать удаленно, не копируя дампы между серверами:

```
pg_restore -d mydatabase -j 2 dumpfile
```

И еще

Выполняя восстановление из резервной копии произвольного формата (-F c), можно также использовать опцию -l, чтобы вывести список содержимого архива. После этого вы сможете отредактировать файл, удаляя, помещая в комментарии, меняя порядок действий. Команда `pg_restore` с опцией -L может использовать файл листинга в качестве входного.

Восстановление удаленной или поврежденной базы данных

Иногда требуется восстановить базу данных полностью – это гораздо проще, чем восстанавливать отдельную таблицу. Многие пользователи помещают все таблицы в одну базу данных; для таких случаев советы из данного раздела не пригодятся.

Как это сделать

Способ зависит от того, какого типа резервная копия у вас есть. Если есть копии разных типов, то возможны варианты.

Логическое (из резервной копии произвольного формата, созданной с помощью команды `dump -F c`).

Восстановите базу данных на целевом сервере. Это можно сделать удаленно, не копируя дампы между серверами.

```
pg_restore -d myfreshdb -j 4 dumpfile
```

Логическое (из скриптового дампа, созданного утилитой `pg_dump`).

Восстановите базу данных на целевом сервере. Это можно сделать удаленно, не копируя дампы между серверами.

```
psql -f myscriptdump.sql myfreshdb
```

Логическое (из скриптового дампа, созданного утилитой `pg_dumpall`).

Не существует простого способа извлечь необходимые таблицы из скриптового дампа. Нужно использовать следующую процедуру:

1. Найти подходящий сервер или создать виртуальный.
2. Выполнить скрипт полностью:

```
psql -f myscriptdump.sql
```
3. Когда восстановление завершится, можно создать дамп таблиц, следуя инструкциям раздела «Горячее логическое резервное копирование одной базы данных».
4. Воссоздать базу данных на целевом сервере, как было описано выше в этом разделе (применительно к логическим резервным копиям).

Физическое.

Чтобы восстановить одну базу данных из физической резервной копии, необходимо:

1. Найти подходящий сервер или создать виртуальный.
2. Полностью восстановить сервер включая все базы данных и все таблицы, как показано в предыдущих разделах, посвященных физическому восстановлению.
3. Если вы хотите остановить восстановление на определенной временной точке, найдите соответствующий раздел в конце этой главы.
4. На восстановленном сервере создайте дамп таблиц базы данных, следуя инструкциям раздела «Горячее логическое резервное копирование одной базы данных».
5. Воссоздайте базу данных, как было показано выше в данном разделе (применительно к логическим резервным копиям).

Улучшение производительности резервного копирования и восстановления

Производительность имеет значение в любой базе данных среднего или большого размера.

Производительность резервного копирования часто становится серьезной проблемой, так как выделяемые для этого ресурсы системы ограничены. Может быть наложено ограничение на длительность копирования, например если оно производится каждое воскресенье.

Производительность восстановления порой может оказаться еще важнее, чем производительность резервного копирования.

Подготовка

- **Физическое резервное копирование:** улучшение производительности физического резервного копирования может быть достигнуто параллельным выполнением – то есть копированием файлов в нескольких задачах. Чем больше задач вы используете, тем большая часть ресурсов системы задействована. При резервном копировании можно пропустить некоторые файлы:
 - все файлы, созданные администратором базы данных, которых не должно быть;
 - все файлы каталога `pg_xlog`;
 - все устаревшие файлы логов сервера в каталоге `pg_log` (даже текущий).

Помните, что безопаснее ничего не исключать, поскольку если вы пропустите что-нибудь критически важное, то будут утеряны данные. Также помните, что узким местом при резервном копировании могут быть дисковая подсистема или сетевое подключение. В больших системах для целей резервного копирования выделяют специальные локальные сети.

- **Логическое резервное копирование:** в предыдущем рецепте говорилось, что для резервного копирования всех баз данных на сервере следует использовать утилиту `pg_dump` параллельно в нескольких задачах. Если вы хотите ускорить работу отдельного процесса `pg_dump`, к сожалению, придется констатировать факт: пока не существует действенного простого способа сделать это. Если вы используете сжатие, просмотрите примечания в конце данного раздела.
- **Физическое восстановление:** как и при физическом резервном копировании, процесс завершится быстрее, если использовать многозадачность.
- **Логическое восстановление:** используете ли вы `psql` или `pg_restore`, можно увеличить скорость выполнения, присвоив параметру `maintenance_work_mem` значение 128MB или больше в файле `postgresql.conf`, либо для того пользователя, который проводит восстановление. Если ни тот, ни другой способ непригоден, можно установить опцию в переменной окружения `PGOPTIONS` следующим образом:

```
export PGOPTIONS = "-c work_mem = 128000"
```

для последующих подключений к базе данных.

Если вы используете архивирование или потоковую репликацию, то запись в логи транзакций может вызвать проблемы. Установите значение параметра `wal_buffers` между 2000 и 10000, а значение параметра `checkpoint_segments` сделайте равным 1024, чтобы снизить нагрузку.

Если вы не используете архивирование или потоковую репликацию или можете отключить их на время восстановления, то есть смысл уменьшить количество операций записи в логи транзакций. Еще один вариант – использовать опцию `-single-transaction`: это также поможет улучшить производительность.

Если резервная копия делалась с помощью команды `pgdump -F` в произвольном формате, восстановление можно распараллелить:

```
pg_restore -j КоличествоЗаданий
```

Следует аккуратно выбирать степень параллельности. Прежде всего надо ориентироваться на количество процессоров в системе. Не забывайте, что объем оперативной памяти ограничен. Каждое задание использует вплоть до `maintenance_work_mem`, так что, когда дело дойдет до больших индексов, часть процессов может быть вытеснена в файл подкачки. Следите, чтобы значения параметров `shared_buffers` и `maintenance_work_mem` и количество заданий соответствовали друг другу.

Не забудьте выполнить `ANALYZE` для всех созданных объектов. Это произойдет само собой, если включена автоматическая сборка мусора. Полное отключение процесса `autovacuum` может улучшить производительность восстановления – не забудьте потом включить его снова, иначе при повторных запусках приложения могут показать крайне низкую производительность.

Как это работает

Выполняйте физическое резервное копирование и восстановление так, как вам удобно. Копируйте файлы настолько быстро, насколько пожелаете, каким угодно способом. Верните их на место при восстановлении тем же способом или любым другим.

Логическое резервное копирование и восстановление включают в себя перемещение данных в базу и из базы. Обычно это происходит медленнее, чем при физическом копировании. Особенно при восстановлении воссоздание индексов и ограничений требует времени, даже если выполняется с использованием многозадачности. Планируйте заранее, измеряйте производительность алгоритмов резервного копирования и восстановления.

И еще

Использование сжатия при резервном копировании считают способом уменьшить объем, необходимый для хранения резервных копий. Даже умеренное сжатие может использовать много процессорного времени. Сжатие также может уменьшить расходы на передачу файлов по сети, поэтому четких правил использования сжатия не существует.

Сжатие файлов WAL при физическом резервном копировании обсуждалось ранее: утилита `pg_lesslog` доступна по адресу http://pgfoundry.org/frs/?group_id=1000310. Физические резервные копии могут быть подвергнуты сжатию разными способами, в зависимости от используемого механизма копирования. По умолчанию дампы произвольного формата подвергается сжатию. В сжатых файлах резервных копий отдельные объекты также доступны.

Создаваемые с использованием опции `--compress` скриптовые дампы сжимаются, как если бы вы сначала создали дампы, а затем сжали его. Доступ к отдельным таблицам невозможен.

Утилиты PostgreSQL имеют опции для сжатия и разворачивания, но это не всегда эффективно. Команда

```
pg_dump --compress=0
```

обычно выполняется медленнее, чем

```
pg_dump | gzip
```

Можете использовать любой быстрый компрессор по своему выбору; результат зависит от типа сжимаемых данных.

Смотрите также

Если резервное копирование слишком накладно; приемлемым решением может послужить резервное копирование реплики. Просмотрите рецепты в главе 12, чтобы узнать, как создать реплику.

Инкрементальное и дифференциальное резервное копирование и восстановление

При проблемах с производительностью резервного копирования в большой базе данных PostgreSQL можно задуматься об использовании инкрементального или дифференциального резервного копирования.

При инкрементальном резервном копировании копируются все файлы, измененные после последнего полного резервного копирования. Для восстановления необходимо иметь полную резервную копию и все наборы инкрементальных изменений.

При дифференциальном резервном копировании сохраняются все индивидуальные изменения после последнего полного копирования. При восстановлении нужно сначала восстановить полную резервную копию, а потом применить все последующие изменения.

Как это сделать

Чтобы выполнить дифференциальное физическое резервное копирование, можно использовать команду `rsync` для сравнения существующих файлов с предыдущей полной копией, а затем восстановить только измененные блоки данных. Переписывать последнюю резервную копию новой неправильно – копий должно быть две или больше. Примерный график резервного копирования выглядит так:

Таблица 11.2.

День недели	Резервная копия, набор 1	Резервная копия, набор 2
Воскресенье	Новая полная копия в набор 1	Новая полная копия в набор 2
Понедельник	Изменения для набора 1	Изменения для набора 2
Вторник	Изменения для набора 1	Изменения для набора 2
Среда	Изменения для набора 1	Изменения для набора 2
Четверг	Изменения для набора 1	Изменения для набора 2
Пятница	Изменения для набора 1	Изменения для набора 2
Суббота	Изменения для набора 1	Изменения для набора 2

Следует хранить хотя бы два полных набора.

Во многих больших базах данных есть таблицы только для добавления записей. В таком случае легко копировать части этих таблиц. Если добавляемые записи могут быть идентифицированы по дате или подобному полю, – еще проще. В любом случае нужен надежный способ пометки, где что находится в резервной копии.

В общем случае нет простого способа выполнять дифференциальное резервное копирование с помощью утилиты `pg_dump`.

Как это работает

В базах данных PostgreSQL не ведется явный учет дат последних изменений или подобной информации о таблицах. Таблицы PostgreSQL

хранятся на диске в виде файлов, поэтому можно попробовать использовать *время модификации* (mtime) этих файлов, хранимое файловой системой. Если по каким-то причинам оно недостоверно, лучше не прибегать к инкрементальному резервному копированию.

Утилита `pg_dump` не позволяет использовать оператор `WHERE`, поэтому, даже если в таблице есть поле для пометки времени последнего изменения, использовать его можно только вручную.

И еще

Полезную информацию о ротации резервных копий можно найти по адресу: http://en.wikipedia.org/wiki/Backup_rotation_scheme

Что касается инкрементального резервного копирования, следует заметить, что методы репликации работают, непрерывно применяя изменения к полной копии. Это можно рассматривать как инкрементальную обновляемую резервную копию. Также это называют стратегией «вечно инкрементального» резервного копирования. Изменения применяются заранее, поэтому восстановление должно быть простым и быстрым. Можно также производить и резервное копирование, но делать его на реплике.

Можно создать утилиту, которая делает дифференциальное резервное копирование блоков данных. Можно читать каждый блок данных и проверять его регистрационный номер (Log Sequence Number, LSN), чтобы проверить, изменился ли блок после предыдущего копирования.

Интересным проектом является `pg_rman`; информация о нем доступна по адресу: <http://code.google.com/p/pg-rman/>

Утилита `pg_rman` читает измененные блоки данных и сжимает их, используя детальную информацию о внутреннем устройстве блоков данных PostgreSQL. Существующие ошибки могут вызвать потерю данных в резервных копиях. В основном проекте PostgreSQL утилита пока считается проблемной, так что использовать ее без договора о технической поддержке нежелательно. Такую поддержку предоставляют различные компании – обратитесь к ним.

В версии 1.1.2 `pg_rman` определенно производит меньшие резервные копии, но пока не позволяет распараллеливать процесс. В результате полное или инкрементальное резервное копирование с использованием параллельных процессов может быть более производительным.

На проект `pg_rman` определенно следует обращать внимание в будущем.



ГЛАВА 12.

Репликация и обновления

В этой главе рассматриваются следующие темы:

- Репликация: основные термины
- Лучшие способы репликации
- Файловая репликация с передачей журналов
- Настройка потоковой журнальной репликации
- Управление репликацией с передачей журналов
- Управление режимом горячего резервирования
- Выборочная репликация с помощью Londiste 3.0
- Выборочная репликация с помощью Slony 2.0
- Балансировка нагрузки с помощью pgpool-II 3.0
- Обновление до нового минорного выпуска (например, с 9.0.0 до 9.0.1)
- Мажорные обновления на месте (например, с 8.4 до 9.0 или с 9.0 до 9.1)
- Мажорные онлайн-обновления с помощью инструментов репликации

В репликации нет волшебства, хотя с ее помощью можно делать удивительные вещи. Лучшая репликация – та, что просто работает. О том, как этого добиться, и рассказывается в данной главе.

Репликация потребует от вас понимания, усилий и терпения. При решении многих вопросов важно сделать правильный выбор. Предпочтение отдавайте простым подходам и способам, обеспечивающим надежность работы.

Репликация в несложных формах появилась уже в версии 8.2 PostgreSQL и постоянно улучшалась. Сторонние проекты и инструменты всегда играли важную роль в развитии PostgreSQL, большинство их написано и поддерживается очень сильными разработчиками. Пессимисты считают, что сторонние влияния ослабляют PostgreSQL и подчеркивают недочеты. Однако PostgreSQL работает с множеством инструментов репликации, и вы можете выбирать из массы вариантов нужный для решения конкретной практической задачи. С данной точки зрения и написана эта глава – примерно половина ее посвящена инструментам, не являющимся частью ядра PostgreSQL.

Все упомянутые здесь инструменты поддерживаются и активно улучшаются основными разработчиками PostgreSQL. Изменения в данной области происходят быстро, и некоторые описываемые ниже ограничения могут быть устранены на тот момент, когда вы будете читать эту книгу. Сверяйтесь с документацией каждого инструмента или проекта.

Что выбрать? Этот вопрос повторяется много раз. Ответ зависит от конкретной ситуации. Часто люди используют какую-то технику на одном сервере, а на других серверах применяют иной способ. Даже разработчики выбирают чужие инструменты, если они больше подходят для решения задачи. Каждый из описанных в книге инструментов и методов рассматривается в контексте, в тех или иных обстоятельствах. Если что-то здесь не упоминается, то возможно, что это не лучшее решение по каким-то причинам, а некоторые инструменты и методы просто не рекомендуется использовать, потому что они недостаточно развиты.

Репликация: основные термины

Технология репликации может показаться очень сложной. Можно понять тех, кто считает, что так и задумано. Существует много разновидностей, у каждой свои сторонники, а преимущества и недостатки горячо обсуждаются.

В основе репликации лежат простые принципы, зная которые легко работать с различными ее вариантами. В данной книге репликация описывается максимально беспристрастно, по возможности с использованием стандартных терминов.

Как это работает

Репликация базы данных – это технология хранения копий данных на удаленной системе.

Вот две основные причины, по которым следует воспользоваться репликацией:

- высокая доступность. Удаленный сотрудник всегда может получить доступ к данным через их копии на нескольких системах;
- перемещение данных. Данные используются дополнительными приложениями или на другом оборудовании. Например, при управлении справочными данными (Reference Data Management) один центральный сервер может пересылать

информацию многим другим приложениям, системам бизнес-анализа и деловой отчетности (Business Intelligence/Reporting Systems).

Каждая из представленных тем сложна, существует много архитектур и разновидностей для решения этих задач.

В книге рассматривается перемещение данных, при котором не происходит обработки информации, только копирование с одного сервера PostgreSQL на другой. Описание инструментов *ETL* (Extract, Transform, Load – дословно «извлечение, преобразование, загрузка»), *EAI* (Enterprise Application Integration, интеграция информационных систем), миграции между базами данных, стратегии организации хранилищ данных выходит за рамки этой книги. Темы, безусловно, интересные, но здесь они не будут освещены.

Рассмотрим основы архитектуры. Обычно отдельные серверы БД называют *узлами*. Группу серверов, объединенных репликацией, именуют *кластером*. Термин обычно расшифровывается именно так, но в PostgreSQL он имеет еще два совершенно отдельных значения. Во-первых, иногда кластером называют совокупность баз данных на одном сервере, хотя предпочтительнее термин «сервер баз данных». Во-вторых, команда `CLUSTER` используется для упорядочения строк в таблице особым образом.

Первый сервер баз данных называют также *главным* (master server), *первичным* (primary server), *обеспечивающим* (provider server), *отправителем* (sender server), *источником* (source server).

Второй сервер баз данных называют *резервным* (standby), *подчиненным* (slave), *подписчиком* (subscriber) или *получателем* (receiver). Резервных серверов может быть несколько.

Все описываемые здесь системы репликации используют один главный и несколько резервных серверов. Системы с несколькими главными серверами обсуждаются кратко.

Главный сервер и резервный сервер являются ролями, которые в разное время могут выполнять разные узлы. Чтобы передать роль главного сервера другому узлу, выполняется процедура *переключения* (switchover). Если происходит отказ главного сервера, и он не может восстановиться, то происходит более жесткое *переключение при отказе* (failover). Процедуры во многом схожие, но удобнее использовать для их обозначения разные термины.

Программное обеспечение, контролирующее кластер и в некоторых случаях автоматически инициирующее переключение при отказе, иногда называют *кластерным программным обеспечением*

(clusterware). Оно может также выполнять другие функции, например балансировку нагрузки.

Ключевым аспектом репликации является то, что изменения данных происходят на главном сервере и затем передаются на резервные узлы. В некоторых случаях резервный сервер может передавать изменения другим резервным серверам – этот процесс называют *ретрансляцией (relay)*.

Время, за которое изменения передаются на резервные серверы после фиксации транзакции на главном сервере, имеет большое значение. Обычно его называют *латентностью*, или *задержкой репликации*. Задержку удобно измерять в секундах. Изменения должны быть применены на резервном сервере, затрачиваемое на это время называют *задержкой применения (apply delay)*. Изменения данных часто пересылаются пакетами. Увеличение размера пакетов может повысить эффективность пересылки данных, но при этом увеличиваются задержка репликации и задержка применения. Полное время, за которое запись копируется с главного сервера на резервный, равно сумме задержки репликации и задержки применения. Некоторые авторы описывают эти характеристики иначе и иногда смешивают одну с другой. В отдельных случаях задержку применения измеряют в байтах (чаще мегабайтах) еще не примененных изменений. Пропускная способность, или скорость передачи, измеряемая в мегабайтах, – интересная характеристика, но к латентности (задержке репликации) имеет отдаленное отношение.

О синхронной репликации говорят в том случае, если изменения данных подтверждаются как отосланные с главного сервера на резервный до того, как подтверждена фиксация транзакции. Если изменения пересылаются после фиксации транзакции, говорят об асинхронной репликации. При *синхронной репликации* задержка напрямую влияет на производительность главного сервера. При *асинхронной репликации* главный сервер работает на полную мощность, хотя возникает риск, что резервный сервер будет не поспевать за главным. За асинхронной репликацией следует наблюдать на случай значительного увеличения лага.

Все формы репликации «главный/резервный» инициируются примерно по одному образцу. Включается захват изменений, и полная реплика данных посылается на резервный сервер. После этого начинается применение изменений. Задержка репликации сразу после начального копирования равна времени копирования. Потом резервный сервер начинает наверстывать отставание, и задержка репликации

становится меньше. Это называется *периодом выравнивания* (catch up period). Под высокой нагрузкой главный сервер производит много изменений, и выравнивание длится дольше, а в некоторых случаях неприемлемо долго. Учтите это при наблюдении и планировании. Оптимальные параметры репликации могут быть разными в период выравнивания и во время нормальной работы.

Репликация копирует все таблицы или какую-либо их часть (*избирательная репликация*). При использовании избирательной репликации можно группировать выбранные объекты в репликационный набор. Если вы предпочитаете избирательную репликацию, помните, что затраты на управление возрастают с увеличением количества объектов. Нужно предусмотреть дополнительное время на административные задачи, особенно если управлять приходится сотнями объектов.

И еще

Архитектуры со многими главными серверами порождают некоторые проблемы, которые сложно решить для общего случая. Классическая статья Джима Грея (Jim Gray) об архитектурах со многими главными серверами – «The Dangers of Replication and a Solution» (Опасности репликации и решение) – считается наилучшим объяснением проблем, возникающих при использовании многих главных серверов. Вот цитата из аннотации: «Репликация данных на многие узлы, когда источников изменений несколько, проблематична». Также прямо заявлена основная проблема: «Репликация, при которой изменения идут откуда угодно, когда угодно, куда угодно, работает нестабильно при увеличении нагрузки».

Статью «The Dangers of Replication and a Solution» вы легко найдете в Интернете по ее названию.

Личный опыт авторов в использовании кластерных решений с несколькими главными серверами не был легким. Операционная нестабильность неприятна в транзакционных системах баз данных, ее бывает трудно предсказать и исправить. Как правило, к применению подобных архитектур стремятся те, кто еще не пробовал с ними работать. Практические трудности могут быть со временем решены, но, по мнению авторов, программному обеспечению, используемому в настоящее время, до этого далеко.

Схемы репликации с одним главным сервером не только просты, но и надежны. Они хорошо выдерживают высокие нагрузки, не ставят

особых требований к дизайну приложений и подходят для решения обычных задач.

Статья Джима Грэя описывает так называемый *шардинг* (sharding). *Горизонтальное масштабирование без совместного использования ресурсов* (shared nothing scale-out) иногда помогает, но не является панацеей. Многим людям эти возможности не нужны, поэтому в данной книге они почти не описываются. PL/Proху специально спроектирован для шардинга в PostgreSQL. Однако детальное рассмотрение выходит за рамки этой книги. Разумеется, придется обойти молчанием часть проектов, связанных с репликацией и кластерингом в PostgreSQL. Здесь освещены только основные схемы репликации с одним главным сервером.

Смотрите также

Дополнительную информацию можно найти по адресу:

[http://en.wikipedia.org/wiki/Replication_\(computer_science\)](http://en.wikipedia.org/wiki/Replication_(computer_science))

Лучшие способы репликации

Наиболее удобные способы репликации описаны в этой главе.

Как это сделать

- Используйте одинаковое оборудование и ОС на всех серверах. Репликация позволяет изменять роли узлов. Если происходит переключение или переключение при отказе на систему другой конфигурации, могут возникнуть проблемы с производительностью, и трудно будет обеспечить бесперебойную работу системы.
- По возможности конфигурируйте все системы одинаково. Используйте одинаковые точки монтирования, одинаковые имена каталогов, одинаковые имена пользователей – сделайте одинаковым все, что только можно. Не поддавайтесь искушению сделать одну систему более важной, чем другие. Различия могут стать причиной отказов и только усложнят дело.
- Не давайте системам имена, вводящие в заблуждение. Не называйте одну систему Master, а другую – Slave. При переключении ролей это может помешать. Давайте имена, не связанные с ролями. Роли меняются, имена не должны меняться. Если

отказывает одна система и вы вводите в строй новую систему, не давайте ей имя старой; выдумайте новое имя. Не выбирайте имена, связанные с бизнесом. Не используйте имена, связанные с состоянием систем, чтобы не звучали фразы типа «красный сигнал на сервере Желтый». Не давайте серверам имена городов, если серверы могут находиться в разных городах, иначе со временем Лондон может оказаться в Эдинбурге, а Париж в Риме.

- Поддерживайте синхронизацию системных часов. Если вы просматриваете логи с нескольких серверов, лучше быть полностью уверенным, к какому времени относятся записи.
- Используйте одну временную зону без сложных правил. Используйте универсальное время (Coordinated Universal Time, UTC) или похожую зону. Не применяйте временные зоны с летним временем. Это приведет к ошибкам, так как серверы могут находиться в разных часовых поясах и разница во времени может изменяться в течение года. Даже если все ваши серверы находятся в пределах одного часового пояса, позже могут добавиться серверы в других часовых поясах.
- Наблюдайте за всеми серверами. Для высокой доступности данных необходимо регулярно проверять работоспособность серверов. Тем, кто считает, что настройка репликации – это разовая сделка, предлагаю думать о репликации скорее как о долгосрочном договоре, который нужно поддерживать, чтобы он был выгодным.
- Наблюдайте за задержкой репликации между серверами. Все формы репликации требуют корректной передачи данных. Необходимо знать, за какой срок информация будет передана с одного сервера на другой, чтобы понимать, правильно ли выполняется репликация. Данные могут пересылаться «всплесками», и вы должны следить, чтобы колебания объемов информации оставались в разумных пределах.

Важно помнить, что при асинхронной репликации объем возможных потерь данных связан с величиной задержки репликации, но не с задержкой применения. Если придется выбирать, то большая задержка применения более приемлема, чем большая задержка репликации.

Ранее уже упоминалось, что при инициализации репликации задержка велика и со временем уменьшается. Для больших баз данных стабилизация задержки может занять несколько дней. Не забывайте наблюдать за системой в этот период.

Файловая репликация с передачей журналов

Доставка журналов (*log-shipping*) является способом репликации, поддерживаемой многими системами управления БД. На главном сервере изменения в базах данных записываются в лог транзакций, файл пересылается на резервный сервер, а там изменения, содержащиеся в логе, выполняются.

Файловая репликация на основе доставки журналов появилась в версии 8.2 сервера PostgreSQL, то есть за четыре с небольшим года до выпуска версии 9.0. Эта форма проста, надежна и требует мало ресурсов.

Эта техника по большей части заменена потоковой репликацией в версии 9.0, но ее все еще можно с удобством использовать как часть стратегии резервного копирования. Полезно разобраться, как это работает, поскольку она может применяться в начальной фазе настройки потоковой репликации в большой системе.

Подготовка

Если вы пропустили начало главы, прочтите эти рецепты сейчас. Репликация слишком сложна, и даже если вы думаете, что знаете, о чем речь, не помешает сверить свои знания с основными понятиями и терминами, которые здесь употребляются. Заметьте, что в контексте репликации с доставкой журналов главный узел называется *первичным узлом* и эти термины равноправны.

Как это сделать

Прodelайте следующие шаги для начальной конфигурации файловой доставки журналов:

1. Уточните, где находится ваш архив и достаточно ли места для хранения. В этом рецепте предполагается, что архив расположен на резервном узле в каталоге, имя которого хранится в переменной окружения `$PGARCHIVE`. Это фиксируется в настройках главного и резервного серверов; главный сервер пишет в данный каталог, а резервный читает из каталога. Имя резервного сервера хранится на главном сервере в переменной `$STANBYNODE`.

2. Проверьте настройки безопасности. Серверы должны обмениваться ключами, чтобы на них можно было выполнять команду `rsync` в обоих направлениях (смотрите один из следующих шагов).
3. Измените на главном сервере параметры в файле `postgresql.conf` следующим образом:

```
wal_level = 'archive'
archive_mode = on
archive_command = <scp %p $STANDBYNODE:$PGARCHIVE/%f>
archive_timeout = 30
```
4. При необходимости измените параметры режима горячего резервирования (смотрите один из следующих рецептов).
5. Сделайте резервную копию – этот процесс очень схож с описанным в одиннадцатой главе процессом физического копирования:
 - начните резервное копирование следующей командой:

```
psql -c 'select pg_start_backup('base backup for log shipping')'
```
 - скопируйте файлы данных (за исключением каталога `pg_xlog`). Потребуется выполнить некоторые настройки безопасности, чтобы команда `rsync` могла выполняться без ввода пароля. Если вы пропустили шаг 2, выполните следующее:

```
rsync -cva --inplace --exclude=*pg_xlog* \
${PGDATA}/ $STANDBYNODE:$PGDATA
```
 - остановите резервное копирование, выполнив следующее:

```
psql -c 'select pg_stop_backup(), current_timestamp'
```
6. Измените параметры в файле `recovery.conf` на резервном сервере:

```
standby_mode = 'on'
restore_command = <cp $PGARCHIVE/%f %p>
archive_cleanup_command = <pg_archivecleanup $PGARCHIVE %r>
trigger_file = </tmp/postgresql.trigger.5432>
```
7. Запустите резервный сервер.
8. Внимательно наблюдайте за задержкой репликации в течение периода выравнивания. В это время задержка репликации гораздо больше, чем в обычное время. Рекомендуется отключить режим горячего резервирования только на время начального периода.

Не выполняйте команды вручную – напишите скрипт, даже если вы занимаетесь тестированием или просто исследуете возможности. Если что-то пойдет не так, вы захотите быстро все запустить сначала, и набор команд вручную не только потребует усилий, но и явится дополнительным источником ошибок.

Как это работает

Файлы лога транзакции (WAL) пишутся на главном сервере. Если установить параметр `wal_level`, то собираться будут все данные, и WAL не будет оптимизироваться. Файлы WAL посылаются с главного сервера командой `archive_command`, и резервный сервер читает файлы WAL с помощью команды `restore_command`, а затем применяет изменения.

Файл посылается после его достижения полного размера, или когда проходит время `archive_timeout` (в секундах) с тех пор, как кто-либо из пользователей выполнял запись в лог транзакции. Если сервер долго не пишет новых данных в лог транзакции, то файлы сменяются каждые `checkpoint_timeout` секунд – это нормально и не представляет проблемы.

Поскольку архив находится на резервном сервере, `restore_command` – обычная команда копирования (`cp`). Если бы архив находился на третьей системе, надо было бы либо монтировать удаленную файловую систему, либо использовать команду для копирования по сети.

Команда `archive_cleanup_command` обеспечивает хранение в архиве только тех файлов, которые необходимы резервному серверу на тот случай, если его придется остановить или перезагрузить. Более старые файлы удаляются, чтобы архив не переполнился. За количеством файлов в архиве также следует следить.

В приведенной конфигурации модуль `contrib/pg_archivecleanup` используется для удаления файлов из архива. Он входит в состав PostgreSQL с версии 9.0. Его предназначение – работа с одним резервным сервером в каждый момент времени. Заметьте, что у команды `pg_archivecleanup` два параметра: имя файла и строка `%f` с пробелом между ними. PostgreSQL преобразует `%f` в имя файла.

Если используется несколько резервных серверов, то общий архив является «узким» местом, возможной причиной отказа, чего следует избегать, поэтому на каждом резервном сервере должен быть свой архив. Впишите команду `archive_command` в скрипт, чтобы выполнять архивирование в несколько мест:

```
archive_command = 'myarchivescript %p %f'
```

и вы сможете написать `myarchivescript` примерно следующим образом (но нужно добавить обработку ошибок на тот случай, если резервные серверы выключены):

```
scp $1 $STANDBYNODE1:$PGARCHIVE/$2
scp $1 $STANDBYNODE2:$PGARCHIVE/$2
scp $1 $STANDBYNODE3:$PGARCHIVE/$2
```

Начальное копирование, или создание базовой резервной копии, выполняется при помощи утилиты `rsync`, для чего может потребоваться прямая безопасная авторизация, например с использованием SSH и обмена ключами. Вы можете выбрать другой способ для создания базовой резервной копии.

И еще

Наблюдать за файловой доставкой журналов можно несколькими способами, например смотреть на текущие файлы на главном и резервном серверах. Вот как это происходит:

```
ps -ef | grep archiver                                on master
postgres: archiver process last was
00000001000000000000000040
ps -ef | grep startup                                  on standby
postgres: startup process waiting for
00000001000000000000000041
```

До версии 9.0 было трудно точно измерить задержку репликации без импровизаций, однако такие методы в книге не рассматриваются. Мониторинг репликации будет описан чуть позже.

Некоторые из советов по сжатию файлов WAL из одиннадцатой главы пригодятся и здесь, хотя в основном файлы WAL не нужно хранить так долго, чтобы их объем становился проблемой. В этом случае сжатие файлов только замедлит работу.

Смотрите также

Инструкции по настройке для PostgreSQL версий от 8.2 до 8.4 действительно почти полностью для версии 9.0 и последующих. Различие состоит в том, что понадобится указать параметр `wal_level`. Запомните, что описываемые здесь процедуры не являются стандартной конфигурацией и могут отличаться от ранних версий. В версии 9.0 и последующих утилита `pg_Standby` уже не является необходимой, так как сервер выполняет многие ее функции. Если вы предпочтете использовать утилиту `pg_Standby` с PostgreSQL 9.x, то вам не понадобятся параметры `archive_cleanup_command`, `Standby_mode` и `trigger_file`.

Улучшенная версия утилиты `pg_standby` является частью проекта `2warm`.

Об этом наборе инструментов можно прочитать подробнее по адресу <http://projects.2ndQuadrant.com/2warm>.

Настройка потоковой журнальной репликации

Доставка журналов является разновидностью репликации, используемой во многих системах управления базами данных. Файлы лога транзакции (WAL) пишутся на главном сервере, посылаются с главного сервера на резервный, и там изменения применяются.

Потоковая журнальная репликация появилась в версии 9.0 PostgreSQL. Ко времени выпуска эта технология уже примерно три года проектировалась, разрабатывалась и тестировалась. Основная ее особенность состоит в том, что данные передаются напрямую с главного сервера на резервный, обеспечивая интегрированную безопасность и уменьшение задержки репликации.

Есть два основных способа использования потоковой репликации: с дополнительным архивом и без него. Здесь описана разновидность без дополнительного архива, как наиболее простая и эффективная. Однако такой подход не годится для больших баз данных, а почему — читайте ниже.

Подготовка

Если вы не прочли разделы этой главы «Репликация: основные термины» и «Лучшие способы репликации», ознакомьтесь с ними сейчас. Заметьте, что в контексте потоковой репликации главный узел называется первичным и эти термины равноправны.

Как это сделать

Выполните следующие шаги:

- Определите главный и резервный узлы, убедитесь, что они сконфигурированы в соответствии с рецептом «Лучшие способы репликации».
- Проверьте настройки безопасности. Создайте на главном сервере учетную запись пользователя репликации, если ее еще нет:

```
CREATE USER repuser
SUPERUSER
LOGIN
CONNECTION LIMIT 1
ENCRYPTED PASSWORD 'changeme';
```

- Сделайте так, чтобы пользователь репликации смог авторизоваться. Следующий пример позволяет получить доступ с любого IP-адреса с применением авторизации по паролю с шифрованием; можно еще больше ограничить доступ. Добавьте следующую строку в файл `pg_hba.conf`:

```
host      replication  repuser      127.0.0.1/0    md5
```

- Установите опции логирования на главном и резервном серверах таким образом, чтобы получать более подробную информацию о попытках установить соединение для репликации и о случаях отказа:

```
log_connections = on
```

- Установите параметр `max_wal_senders` в файле `postgresql.conf` или увеличьте его значение, если оно не равно нулю:

```
max_wal_senders = 1
wal_mode = 'archive'
archive_mode = on
archive_command = 'cd .'
```

- Подберите значение для параметра `wal_keep_segments` в файле `postgresql.conf` на главном сервере. Не делайте его большим, чем количество 16-мегабайтных сегментов свободного пространства на устройстве, где находится каталог `pg_xlog`. Если под каталог `pg_xlog` не выделено отдельное устройство, то помните, что не все свободное пространство может быть использовано для файлов логов транзакции.

```
wal_keep_segments = 10000          # e.g. 160 GB
```

- Скорректируйте параметры режима горячего резервирования, если потребуется (смотрите в рецепте ниже).
- Сделайте базовую резервную копию так же, как при физическом резервном копировании (см. одиннадцатую главу):

- начните резервное копирование:

```
psql -c "select pg_start_backup('base backup for
streaming rep')"
```

- скопируйте файлы данных (за исключением каталога `pg_xlog`):

```
rsync -cva --inplace --exclude=*pg_xlog* \
${PGDATA}/ $STANDBYNODE:$PGDATA
```

- остановите копирование:

```
psql -c "select pg_stop_backup(), current_timestamp"
```

- Скорректируйте параметры в файле `recovery.conf` на резервном сервере. Заметьте, что параметр `primary_conninfo` не должен содержать имени базы данных, хотя может содержать любую другую опцию сетевого подключения PostgreSQL. Также заметьте, что все опции в файле `recovery.conf` заключены в кавычки, тогда как для параметров в файле `postgresql.conf` этого не нужно.

```
Standby_mode = 'on'
```

```
primary_conninfo = 'host=192.168.0.1 user=repuser'
```

```
trigger_file = '/tmp/postgresql.trigger.5432'
```

- Запустите резервный сервер.
- Внимательно наблюдайте за задержкой репликации, пока не закончится период выравнивания. В это время величина задержки репликации намного больше, чем ожидается при нормальной работе. Рекомендуется установить `hot_standby = off` только на время начального периода.

Как это работает

Если вы используете несколько резервных серверов, не забудьте скорректировать параметр `max_wal_senders`. Допускается создавать отдельную учетную запись для каждого резервного сервера, но иногда достаточно задать параметр `application_name` в `primary_conninfo`.

Резервные серверы не могут обращаться к другим резервным серверам – только к текущему главному серверу. При попытке подключения к другому главному серверу резервный сервер получит сообщение об ошибке.

Архитектура потоковой репликации такова, что для каждого резервного сервера на главном сервере создается один процесс `WALSender`, а на резервном сервере – процесс `WALReceiver`. Репликация – очень эффективный способ передачи данных, информация в основном передается фрагментами по 8192 байта без буферизации на сетевом уровне.

Процессы `WALSender` и `WALReceiver` будут пересылать и принимать данные, пока очередь не опустеет. Когда данные не передаются, `WALReceiver` «засыпает» на 100 мс, а `WALSender` – на время, установленное в параметре `wal_sender_delay`. Обычно это значение не меняют, так как оно влияет лишь на поведение сервера в периоды бездействия.

Стандартное значение дает хороший баланс между эффективностью и защитой данных. Если соединение между главным сервером и резервным низкоскоростное, а частота записи на главном сервере высока, можно уменьшить значение `wal_sender_delay` до 20–50 мс. Это позволяет снизить потери данных в случае, если главный сервер станет недоступен, но несколько повышает затраты ресурсов на передачу логов резервным серверам.

Резервный сервер соединяется с главным, используя специфичную для PostgreSQL библиотеку `libpq`. Все формы авторизации и безопасного подсоединения применяются при репликации, как и при обычных подключениях. Помните о том, что в сеансе репликации резервный узел является клиентом, а главный – сервером, если будете конфигурировать дополнительные параметры. При стандартном подключении PostgreSQL с помощью `libpq` используются обычные номера портов сетевых служб, поэтому не понадобятся дополнительные правила для брандмауэра. Использование SSL при соединении требует немного больших ресурсов и вызывает задержку репликации.

И еще

Потеря соединения между главным и резервным серверами при непосредственном подключении обнаруживается лишь через некоторое время. Чтобы заметить это как можно быстрее, скорректируйте настройки `keepalive`.

Чтобы резервный сервер обнаруживал пропавшее соединение с главным сервером, задайте опции `keepalives` в параметре `primary_conninfo` в файле `recovery.conf` на резервном сервере:

```
primary_conninfo = '...keepalives_idle= 60 ...'
```

Если вы хотите, чтобы главный сервер заметил, что пропало соединение с резервным сервером потоковой репликации, можно задать параметры `keepalive` в файле `postgresql.conf` на главном сервере примерно так:

```
tcp_keepalives_idle = 60          # time before we send  
keepalives
```

Эта установка будет применяться ко всем соединениям с пользователями и репликационными серверами. Если вы хотите, чтобы опция использовалась только для репликации, то укажите ее следующим образом:

```
primary_conninfo = '...options="-c tcp_keepalives_idle= 60" ...'
```

Во всех предыдущих примерах промежутки времени бездействия соединения, после которого начинают посылаться пакеты `keepalive`,

составляет 60 секунд. Стандартное значение равняется двум часам – его лучше не использовать. Параметров `keepalive` много, и не все они приведены здесь. Здесь нужно упомянуть также опцию `connection_timeout`. Все сложные детали могут быть скрыты в служебном файле соединения, чтобы параметр `primary_conninfo` относился только к одной службе, как описано в первой главе.

Можно также увеличить значение параметра `max_wal_senders`, чтобы восстановить соединение даже раньше, чем будет замечен разрыв. Это позволяет перезапустить соединение вручную, что несколько упрощает работу. Также увеличивается лимит соединений пользователя репликации.

Передача данных может быть остановлена из-за разрыва соединения с резервным сервером или остановки резервного сервера. Если передача данных репликации прекращается по любой причине, происходит попытка возобновления с момента последней передачи.

При потоковой репликации на главном сервере хранятся файлы объемом как минимум равным значению `wal_keep_segments`. Если резервный сервер отключен надолго, на главном сервере не сохранятся данные с момента последней передачи и придется снова настраивать резервный сервер.

Заметьте, что во время получения базовой резервной копии резервный сервер не участвует в репликации, и если процесс длится слишком долго, то репликация не начнется, так как на сервере уже не будет данных за соответствующий период. Вы получите такое сообщение об ошибке:

```
FATAL: requested WAL segment 0000000100000000000000002 has  
already been removed
```

Это очень неприятно, и нет способа исправить ошибку. Поэтому начните с очень большого значения `wal_keep_segments`. Если то же сообщение об ошибке все же появляется, нужно увеличить это значение и попытаться восстановить репликацию. Возможно, следует использовать способы, ускоряющие резервное копирование (см. одиннадцатую главу). Если нельзя увеличить значение `wal_keep_segments`, то для данных репликации требуется устройство большого объема, скорее всего, на другом узле. На главном сервере команда `archive_command` должна помещать файлы на архивный сервер, тогда как выше описана команда-заглушка. Резервный сервер должен будет извлекать из архива файлы с помощью команды `restore_command` и участвовать в репликации с использованием `primary_conninfo`. Таким образом, у главного и резервного серверов будет два режима обмена данными, и они смогут

переключаться между режимами в случае сбоя. Такова типичная конфигурация для больших баз данных. Заметьте, что при этом данные WAL копируются дважды: один раз на архивный сервер, а второй раз – на резервный. Это, конечно, затратно, зато более надежно.

Задайте значение параметра `archive_mode = on`. Чтобы изменить его, потребуется перезапуск, поэтому лучше включить этот режим сразу просто на случай, если он вдруг понадобится. Все, что вам нужно, – команда-заглушка `archive_command`, чтобы проверить, все ли в порядке (ее называют заглушкой потому, что она не выполняет никаких действий, только возвращает код завершения 0).

Можно использовать `archive_command` только до окончания периода выравнивания, затем заменить ее заглушкой и использовать уже только потоковую репликацию. Данные только передаются с главного сервера на резервный, если они записаны на диск (точнее, выгружены из системного буфера дискового обмена). Поэтому присвоение параметру `synchronous_commit` значения `off` не уменьшит задержку репликации, даже если и увеличит производительность главного сервера. Получая данные WAL, резервный сервер выгружает их на диск, чтобы не потерять в случае перезапуска.

Очень важно контролировать потоковую репликацию, как уже отмечалось ранее. Процессы `WALSender` не отражаются в системном представлении `pg_stat_activity`, но за отдельными деталями можно проследить. Используйте следующее представление:

```
CREATE OR REPLACE VIEW pg_stat_replication AS
SELECT
    S.procpid,
    S.usesysid,
    U.rolname AS username,
    S.application_name,
    S.client_addr,
    S.client_port,
    S.backend_start
FROM pg_stat_get_activity(NULL) AS S, pg_authid U
WHERE S.usesysid = U.oid AND S.datid = 0;
```

Управление репликацией с передачей журналов

Неважно, используете ли вы файловый или потоковый способы, – управление кластером репликации с доставкой журналов во многом на них похоже.

Подготовка

Допустим, что вы уже настроили файловую доставку журналов, или потоковую, или ту и другую.

Как это сделать

Мониторинг

При репликации с доставкой журналов важно контролировать процесс. Вы поймете, что лучше всего включить режим горячего резервирования, так как с его помощью проще получить более точную информацию.

Приложения `repmgr` и `pgpool` позволяют наблюдать за репликацией. Дополнения для Munin могут графически отображать задержку репликации и задержку применения.

Может быть, вы захотите самостоятельно вычислить задержку. Для этого запросите текущую позицию на главном и резервном серверах и сравните значения. На главном сервере выполните следующее:

```
SELECT pg_current_xlog_location();
```

На резервном сервере выполните такой запрос:

```
SELECT pg_last_xlog_receive_location();
```

Затем сравните результаты, чтобы оценить задержку репликации. Можно вычислить задержку применения, сравнив предыдущее значение, полученное на главном сервере, со следующим:

```
SELECT pg_last_xlog_apply_location();
```

Сравнивать не слишком просто, так как необходимо получить числа из имени файла и использовать шестнадцатеричную арифметику. Это можно сделать на Perl, Python, Java, С и пр. или, как в следующем примере, на SQL:

```
CREATE OR REPLACE FUNCTION hex2dec(text)
RETURNS bigint LANGUAGE SQL AS
$$
SELECT sum(digit * 16 ^ (length($1)-pos)) ::bigint
FROM (SELECT case
      when digit between '0' and '9' then ascii(digit)
- 48
      when digit between 'A' and 'F' then ascii(digit)
- 55
      end,
      pos as i
FROM (SELECT substring(c from x for 1), x
      FROM (values(upper($1))) as a(c),
      generate_series(1,length($1)) as
t(x))
```

```
        as u(digit, pos)
    ) as v(digit, pos);

$$;
```

Переключение и переключение при отказе

При переключении происходит управляемый переход с главного сервера на резервный. Если все сделать правильно, потери данных не будет. Для безопасности просто завершите работу главного узла. Используйте быстрое выключение либо smart-выключение. Не применяйте немедленное выключение, так как при этом потеря данных почти неизбежна.

Переключение при отказе – это вынужденный переход с главного сервера на резервный из-за аварийного выключения главного сервера. Поэтому на главном сервере не надо ничего делать; будем считать, что его больше нет.

Нужно выбрать один из резервных серверов в качестве нового главного сервера. Переключение резервного сервера в режим главного сервера может быть инициировано созданием файла-триггера, указанного в параметре `trigger_file`. Например:

```
touch /tmp/postgresql.trigger.5432
```

Файл-триггер будет удален после завершения перехода. Заметьте, что резервный сервер станет главным только после применения всех изменений. На это может уйти некоторое время.

Став главным, сервер начнет выполнять все функции, включая архивирование файлов, если это указано в настройках. Убедитесь, что корректны все настройки узла на случай, если он станет главным сервером. Есть вероятность, что они отличаются от настроек первоначального главного сервера.

Прежний главный сервер неожиданно может вернуться в строй; тогда возникнет ситуация, которую называют «разделенный мозг». Позаботьтесь о том, чтобы после передачи «полномочий» прежний главный сервер оставался выключенным.

Автоматическое переключение при отказе, как и управление сложными ситуациями во время переключения при отказе, не обеспечивается стандартными инструментами PostgreSQL. Если узлов много, могут возникать очень сложные ситуации, и для управления ими во многих случаях используют `clusterware`.

Обратное переключение

После переключения с одного сервера на другой может возникнуть необходимость снова переключиться на прежний главный сервер. Это так называемое *обратное переключение* (switchback).

Когда резервный сервер стал главным, он не может снова стать резервным сервером. При репликации с доставкой журналов нет стандартной операции обратного переключения, что многих удивляет. Но недолго найти и обходной путь. Для обратного переключения нужно только выполнить следующее:

1. Перенастроить прежний главный сервер, повторив описанный выше процесс.
2. Переключиться с текущего главного сервера на прежний.

Важно, что если перед первым шагом вы не удалите файлы на прежнем главном сервере, то копирование с помощью `rsync` будет выполнено гораздо быстрее. Если файлы удалены, `rsync` произведет обычное копирование. Если в целевом каталоге присутствуют файлы с такими же именами, как в каталоге-источнике, файлы будут сравнены, а копироваться будут только те, которые отличаются. Поэтому при обратном переключении с использованием `rsync` данных пересылается гораздо меньше, чем при создании новой копии. Возможно, в следующих версиях PostgreSQL эта особенность будет реализована лучше. Есть и способы избежать такой ситуации, например с помощью утилиты `repmgr`, которая будет рассмотрена позже.

Как это работает

Заметьте, что файл-триггер не имеет отношения к репликации, основанной на триггерах. Файл-триггер является механизмом, с помощью которого резервный сервер получает сигнал, что он должен поменять свою роль на роль главного сервера. Имя файла может быть любым, только не забудьте, что оно должно быть разным для разных серверов, если они работают на одном компьютере.

Команда `recovery_end_command` предназначена для очистки при завершении переключения или переключения при отказе. Файл-триггер не требуется удалять вручную, как было рекомендовано в предыдущих версиях.

Смотрите также

Кластерное программное обеспечение может предоставлять дополнительные возможности, такие как автоматическое переключение при отказе, мониторинг, удобное управление репликацией.

`Repmgr` является инструментом с открытым кодом, спроектированным специально для репликации сервера PostgreSQL. Дополнительная

информация доступна по адресу <http://projects.2ndQuadrant.com/repmgr/>.

Continuent Tungsten – коммерческое кластерное приложение, которое поддерживает и PostgreSQL. Его можно приобрести по адресу <http://www.continuent.com/>.

Управление режимом горячего резервирования

Горячим резервированием (hot standby) называется компонент сервера PostgreSQL, позволяющий подключиться к резервному серверу и выполнять запросы без модификации данных. Наиболее важно, что в режиме горячего резервирования можно выполнять запросы в то время, когда резервный сервер продолжает обновляться через файловую или потоковую репликацию.

Режим горячего резервирования позволяет разгрузить главный сервер, переложив часть нагрузки на резервный сервер, а также выполнять запросы во время процесса переключения ролей серверов.

Есть некоторые сложности, с которыми следует разобраться, чтобы успешно использовать режим горячего резервирования, поскольку иногда запросы пользователя конфликтуют с непрерывным применением изменений. Когда новые изменения не передаются, конфликт не возникает.

В связи с режимом горячего резервирования рассматриваются две основные роли. Во-первых, резервный узел обеспечивает замену при отказе первичного узла. Во-вторых, на резервном узле можно выполнять запросы. В некоторых случаях одно мешает другому, и вы должны заранее решить, насколько важна каждая из этих ролей.

Чаще всего роль будущего главного сервера важнее. Хорошо иметь возможность выполнять запросы, но лучше отменить запрос, если он мешает обновлению резервного сервера. Если узлы работают в режиме горячего резервирования, то можно выделить один как будущий главный сервер и выполнять запросы на остальных без оглядки на то, что они являются резервными.

Подготовка

Режим горячего резервирования можно использовать:

- при файловой репликации;
- при потоковой репликации;

- во время восстановления к точке времени;
- на постоянно «замороженном» резервном сервере.

В первых двух случаях нужно настроить репликацию, как описано в предыдущих рецептах. Кроме того, нужно задать следующие параметры:

```
на главном сервере в файле postgresql.conf:
wal_level = 'hot_standby'
на резервном сервере в файле postgresql.conf:
hot_standby = on
```

Эти настройки не являются стандартными. Чтобы изменения вступили в силу, необходимо перезапустить и главный, и резервный серверы.

Новый режим не включается на резервном сервере немедленно, поэтому нужно, чтобы до его перезапуска прошло некоторое время. Этот промежуток времени обычно имеет ту же длительность, что и время `checkpoint_timeout`, а в некоторых случаях больше. Если перезапустить резервный сервер слишком быстро, он не успеет дочитать данные логов транзакции, откажется запускаться и выдаст сообщение `you need to enable Hot Standby` («необходимо включить режим горячего резервирования»). Поэтому будьте терпеливы. Включение режима выполняется при первом перезапуске, и нет необходимости делать это при последующих перезапусках.

Можно задать на резервном сервере в файле `postgresql.conf` следующие параметры:

- `max_standby_archive_delay`
- `max_standby_streaming_delay`
- `vacuum_defer_cleanup_age`
- `trace_recovery_messages`

Постоянно «замороженный» резервный сервер можно создать, задав определенные настройки в файле `recovery.conf`. Параметров `restore_command` и `primary_conninfo` не должно быть, параметру `standby_mode` нужно присвоить значение `on`. В этом режиме сервер запускается, но базы данных остаются в точно таком же состоянии, в котором были, когда была выполнена функция `pg_stop_backup()`.

Следует отметить, что в начальный период выравнивания поддержка репликации намного выше, чем в рабочем режиме. Рекомендуется на данное время отключить режим горячего резервирования. Если пользователи будут подключаться к резервному серверу в этот период, то дополнительное использование ресурсов и возможные конфликты продлят период выравнивания. Когда резервный сервер

поравняется с главным, можно включить режим горячего резервирования и перезапустить сервер.

Как это сделать

Запросы, выполняемые на резервном сервере, обращаются к версии базы данных, которая немного отстает от первичного узла. Это так называемое «согласование в конечном итоге» (eventually consistent). Когда же наступает момент «конечного итога»? Данное время в точности равно сумме задержки репликации и задержки применения, которые обсуждались в разделе «Репликация: основные термины» в начале этой главы. Можно задать верхний предел приемлемой задержки применения, изменяя два параметра: `max_standby_streaming_delay` и `max_standby_archive_delay`.

Чтобы подобрать значения указанных параметров, нужно рассмотреть причины задержки применения – конфликты запросов. Конфликты между главным сервером и запросами на резервном сервере можно разделить на четыре основных типа:

- ресурсы – процессорное время, каналы ввода-вывода и т. д.;
- блокировки – `AccessExclusiveLocks`;
- записи об очистке (`cleanup records`);
- другие особые случаи.

Конфликты из-за ресурсов понять проще всего: когда сервер принимает изменения, меньше ресурсов остается для выполнения запросов. То есть если не получено новых изменений, производительность выполнения запросов выше. Если ожидается отправка изменений с главного сервера, то следует меньше загружать резервный сервер запросами.

Конфликты из-за ресурсов могут замедлять выполнение запросов на резервном сервере, и их можно называть мягкими конфликтами. Конфликты других типов жесткие, из-за них запросы на резервном сервере отменяются или происходит разъединение.

Конфликты из-за блокировок также легко понять: если вы хотите выполнить на главном сервере такую команду, как `ALTER TABLE ... DROP COLUMN`, то таблица должна быть заблокирована от любого доступа. Запрос блокирования посылается и на резервный сервер, и запросы, использующие эту таблицу на резервном сервере, отменяются.

На главном сервере систем с высокой нагрузкой трудно изменить структуру таблиц, поскольку возникает длительное блокирование. Однако таблицы на резервном сервере вполне могут быть доступны

для чтения, пока на главном сервере изменяется их структура. Для этого временно задайте параметры `max_standby_streaming_delay = -1` и `max_standby_archive_delay = -1`, а потом перезагрузите сервер. Когда запись о блокировке будет передана на резервный сервер, дальнейшие изменения будут прекращены. Когда на главном сервере произойдет разблокирование, можно вернуть параметрам их предыдущие значения, что позволит применять изменения.

Использовать для параметра `max_standby` значение `-1` не очень полезно в обычной работе. Ни один пользовательский запрос не будет отменен из-за конфликта с применением изменений, процесс применения будет задержан на неопределенное время. В результате задержка применения может значительно возрасти со временем, в зависимости от частоты и длительности запросов, а также частоты конфликтов. Чтобы найти приемлемое значение для этих параметров, надо больше знать о других типах конфликтов, хотя есть простой способ полностью избежать проблем.

К конфликтам относятся также записи об очистке и особые случаи. Особые случаи бывают редко и/или достаточно просты. Например, таблица удалена на главном сервере, поэтому запросы на резервном сервере отменяются.

Записи об очистке удаляют старые версии кортежей из базы данных, это часть внутренней работы MVCC. Рассмотрим происходящее на главном узле. Команда `VACUUM` удаляет только версии строк, которые недоступны ни одному текущему запросу. Для этого запрашивается список выполняемых запросов, и таким образом гарантируется, что устаревшие версии строк не будут удалены слишком рано. В режиме горячего резервирования главный сервер не получает область видимости для запроса, выполняемого на резервном сервере. В результате становится возможным удаление версий строк, участвующих в запросах на резервном сервере. При их удалении записи об очистке в логах транзакций могут вступить в конфликт с запросами на резервном сервере.

Чтобы обеспечить некоторую защиту от отмены запросов, присвойте параметру `vacuum_defer_cleanup` значение больше нуля. Трудно точно определить правильное значение для этого параметра, но обычно рекомендуется начать с 1000 и затем увеличивать его. Можно предположить, что 1000 соответствует примерно 1 секунде задержки.

Проект `repmgr` предоставляет механизм более точного определения видимости и создавался специально для поддержки режима горячего резервирования в PostgreSQL 9.0. Использование `repmgr`

уменьшает количество запросов, отменяемых из-за конфликтов с записями об очистке, обеспечивая динамическое и более точное определение видимости на резервном сервере.

Если вы хотите полностью «заморозить» резервный сервер, чтобы прекратить применение изменений, то нужно остановить сервер, удалить из файла `recovery.conf` параметры `restore_command` и `primary_conninfo` и перезапустить сервер. Выйти из этого режима можно только в том случае, если в архиве еще будут находиться необходимые файлы WAL, иначе придется заново конфигурировать резервный сервер, начиная с базовой резервной копии.

Если отмены запросов все же происходят, то появляются сообщения об ошибках уровня `error` или `fatal error`. Они помечаются `SQLSTATE 40001 SERIALIZATION FAILURE`. Сообщения обычно обнаруживает приложение, которое затем может повторить запрос.

Как это работает

На резервном сервере изменения читаются из логов транзакций главного сервера и применяются к базам данных резервного сервера. Резервный сервер имитирует выполнение транзакций на главном сервере, чтобы запросы, производимые на резервном, получали информацию о видимости, которая им нужна для того, чтобы полностью соответствовать требованиям MVCC. Поэтому режим горячего резервирования особенно удобен для выполнения множества коротких (быстрых) запросов. Если нагрузка состоит из коротких запросов, то конфликты возникают редко, и сервер работает бесперебойно.

При выполнении более долгих запросов в режиме отчетов (*reporting mode*) нужно изменить настройки или использовать утилиту типа `pergwr` для минимизации конфликтов запросов с записями об очистке.

И еще

Изменения на главном сервере не будут видны на резервном, пока транзакция не зафиксирована. Допустим, есть главный и резервный серверы, задержка репликации составляет 4 секунды. Изменения в ходе транзакции могут занимать около часа. Сколько времени пройдет, прежде чем изменения станут видны на резервном сервере? В режиме горячего резервирования они проявятся на резервном сервере через 4 секунды после фиксации транзакции на главном. Изменения в ходе транзакции на главном сервере транслировались на резервный

сервер, когда транзакция еще не была завершена, и были уже применены, когда с главного сервера пришла запись о фиксации транзакции. Заметьте, что эта ситуация сильно отличается от той, которая возникает при триггерной репликации, например с использованием Slony и Londiste, когда данные об изменениях не передаются, пока транзакция не будет зафиксирована. Таким образом, при триггерной репликации изменения будут видны на резервном сервере лишь спустя несколько минут после фиксации транзакции. Это означает, что при триггерной репликации фактическая задержка применения зависит также от длительности транзакции на главном сервере.

Режим горячего резервирования можно также использовать во время восстановления к точке времени. Файлы WAL не обязаны передаваться с работающего сервера. Нужно только использовать файловое восстановление, а не потоковую репликацию.

Заметьте также, что резервный узел можно выключать и перезапускать обычным образом, используя команды, описанные в предыдущих главах.

Смотрите также

Проект `repmgr` содержит компонент для минимизации конфликтов записей (см. <http://projects.2ndQuadrant.com/repmgr/>).

Выборочная репликация с помощью Londiste 3.0

Londiste является частью пакета программ Skytools, разрабатываемых и используемых компанией Skype для их рабочих транзакционных баз данных. Это репликационная система, построенная на системе общего назначения с очередями событий под названием PgQ (произносится «пи-джи-кью»), модуле расширения PostgreSQL.

Этот инструмент прост, легок и надежен. Наиболее важно то, что Londiste добивается согласованности в конечном итоге между главным и резервным серверами без множества ограничений, и операции на главном сервере свободны от блокировок.

Подготовка

Если вы не прочли разделы «Репликация: основные термины» и «Лучшие способы репликации» в начале этой главы, ознакомьтесь с ними

сейчас. Репликация является сложным предметом, и даже если вы думаете, что знаете, о чем речь, сверьте свои знания с основными понятиями и терминами, которые употребляются здесь.

Как это сделать

Выполните следующие шаги, чтобы настроить репликацию с использованием Londiste:

1. Установите программное обеспечение и создайте каталоги. В системах Linux Debian (Ubuntu) имена программ не имеют суффиксов .py, поэтому напишите просто `pgqadm` и `Longdiste`.
2. Отредактируйте файл конфигурации `clustname_ticker.ini` следующим образом:

```
[pgqadm]
job_name = clustname_ticker
db = dbname=ticker
# how often to run maintenance [seconds]
maint_delay = 600
# how often to check for activity [seconds]
loop_delay = 0.1
logfile = /var/log/londiste/%(job_name)s.log
pidfile = /var/log/londiste/%(job_name)s.pid
```

3. Отредактируйте файл `clustname_londiste.ini` следующим образом:

```
[londiste]
job_name = clustname_londiste
provider_db = dbname=source port=6000 host=groucho
subscriber_db = dbname=target port=6000 host=zeppo
pgq_queue_name = clustname_queue
pgq_lazy_fetch = 500
logfile = /var/log/londiste/%(job_name)s.log
pidfile = /var/run/londiste/%(job_name)s.pid
```

Проверьте соединение с провайдером и подписчиком, указанное в файле `londiste.ini`, для провайдера `provider_db` и подписчика `subscriber_db`.

4. Создайте инфраструктуру базы данных. Выполните следующий код на узле провайдера:

```
pgqadm.py clustname_ticker.ini install
londiste.py clustname_londiste.ini provider install
londiste.py clustname_londiste.ini subscriber install
```

5. Запустите демон тикера на узле провайдера:

```
pgqadm.py clustname_ticker.ini ticker -d
```

6. Запустите демон Londiste на узле подписчика:

```
londiste.py clustname_londiste.ini replay -d
```
7. Определите репликационный набор на узле провайдера. Для репликационного набора из таблиц *a*, *b* и *c* используйте следующее:

```
londiste.py clustname_londiste.ini provider add a b c
```
8. Скопируйте реплицируемые объекты на узел подписчика. Определения должны быть идентичными, а индексы и внешние ключи могут отличаться. Если вы хотите вносить изменения на узле подписчика, напишите скрипт для дополнительных изменений. На узле провайдера выполните следующее:

```
pg_dump --schema-only -t a -t b -t c > clustname_schema.sql
```
9. На узле подписчика используйте следующее:

```
psql -f clustname_schema.sql
```
10. Если вам нужно внести дополнительные изменения на узле подписчика, сделайте это немедленно. Например, у подписчика могут быть совершенно иные индексы для поддержки нагрузки других типов.

```
psql -f clustname_subscriber_changes.sql
```
11. Выполните подписку – на узле подписчика произведите следующее:

```
londiste.py clustname_londiste.ini subscriber add a b c
```
12. Вот и все. Londiste постепенно сократит задержку репликации. За тем, как это происходит, можно проследить:

```
londiste.py clustname_londiste.ini subscriber tables
```
13. Нужно проследить до тех пор, пока все таблицы не будут отображены со статусом OK, или выполнить на узле провайдера:

```
SELECT * FROM pgq.get_consumer_info();
```
14. Потом следует подождать, пока не будут добавлены все внешние ключи к таблицам подписчика, и проконтролировать выполнение следующего запроса на узле подписчика:

```
SELECT count(*) FROM londiste.subscriber_pending_fkeys;
```

Так выполняется настройка выборочной репликации с помощью Londiste.

Если вы хотите добавить в репликационный набор *все таблицы*, следует создать скрипты для выполнения описанных выше шагов для всех таблиц.

1. Выполните шаги 1–6 предыдущей последовательности.
2. Определите репликационный набор на узле провайдера следующим образом:

```
londiste.py clustname_londiste.ini provider add table1
londiste.py clustname_londiste.ini provider add table2
...
londiste.py clustname_londiste.ini provider add tableN
```

3. Скопируйте реплицируемые объекты на узел подписчика. На узле провайдера выполните следующее:

```
pg_dump --schema-only > clustname_schema.sql
```

4. На узле подписчика выполните:

```
psql -f clustname_schema.sql
```

5. Если вам нужно внести дополнительные изменения на узле подписчика, сделайте это немедленно:

```
psql -f clustname_subscriber_changes.sql
```

6. Осуществите подписку – на узле подписчика выполните следующее:

```
londiste.py clustname_londiste.ini subscriber add table1
londiste.py clustname_londiste.ini subscriber add table2
...
londiste.py clustname_londiste.ini subscriber add tableN
```

Как это работает

Все таблицы из одного репликационного набора должны использовать одну очередь. Создать несколько очередей и несколько демонов Londiste, соединяющих одних и тех же провайдера и подписчика, – не проблема, но это явно более сложная схема, которая требуется только в том случае, когда возникают значительные проблемы с производительностью. Для каждой базы данных провайдера нужен только один процесс `pgqadm`, поскольку он предоставляет только тикер.

При добавлении таблиц без явного описания схемы Londiste предполагает, что используется схема с именем `public`. Londiste не использует пути поиска (`search paths`). Если вы добавляете таблицы из другой схемы, имена таблиц должны быть указаны полностью:

```
schemaname.tablename
```

Londiste эффективно использует команду `copy` для перемещения данных, а затем аккуратно применяет изменения, и наборы таблиц в конечном счете совпадут. Нет необходимости включать все таблицы

в один репликационный набор на одной командной строке. Внешние ключи будут восстановлены, когда будет достигнута согласованность.

Согласованность в конечном итоге является, вероятно, самым важным аспектом данной системы репликации. Таблицы копируются отдельно, поэтому нет необходимости блокировать все таблицы на узле провайдера в одной большой транзакции, как это происходит при выполнении `pg_dump`.

Для Londiste при каждом изменении строки добавляется одна строка события в таблице очереди с использованием специальных триггеров. Триггеры устанавливаются автоматически на каждую таблицу командами Londiste. Триггеры, как и многие характеристики Londiste, почти идентичны используемым в Slony.

Измененные строки передаются подписчику пакетами. Пакеты формируются при участии тикера по запросам подписчика, поэтому что если тикер откажет, то пакеты станут очень большими.

У рассматриваемой формы репликации есть определенные недостатки. Во-первых, использование триггеров и таблиц очередей повышает требования к системным ресурсам, но это сильно зависит от типа нагрузки. Следует рассчитывать на 10–20 % накладных расходов. Во-вторых, таблицы очередей занимают место на диске и в оперативной памяти. В-третьих, для добавления, удаления, изменения таблиц в репликационном списке потребуются специальные действия со стороны администратора.

Выгода от использования Londiste состоит в том, что можно сделать выборочную реплику базы данных и не нужно копировать все таблицы.

И еще

Для безопасной остановки Londiste выполните следующее:

```
londiste.py clustname_londiste.ini -stop
```

а в экстренном случае можно сделать следующее:

```
londiste.py clustname_londiste.ini -kill
```

Если вы хотите изменить конфигурацию при включенном сервере, можно перезагрузить конфигурацию:

```
londiste.py clustname_londiste.ini -reload
```

Чтобы проверить состояние, выполните:

```
SELECT queue_name, consumer_name, lag, last_seen  
FROM pgq.get_consumer_info();
```

Поле `lag` позволяет следить за задержкой репликации.

Если репликация остановилась или ее нужно перезапустить, выполните на узле подписчика следующее:

```
londiste.py clustname_londiste.ini repair table1
```

Смотрите также

Необходимую информацию вы найдете на сайте <http://skytools.projects.postgresql.org/doc/>.

Спасибо Марко Крину и Димитрию Фонтейну за дополнительные советы, особенно за http://wiki.postgresql.org/wiki/Londiste_Tutorial – с этой страницы многие начинали свое знакомство с Londiste.

Выборочная репликация с помощью Slony 2.0

Проект Slony – одно из самых давних и наиболее известных средств репликации для PostgreSQL. С его помощью можно делать сложные вещи, но многие считают его трудным в освоении.

Проект Slony1 известен также как Slony-I, так как планировалась разработка Slony-II (правда, пока благие намерения не принесли ощутимых результатов). Первой была выпущена версия 1.0, к настоящему времени вышла версия 2.0. Когда здесь говорится о Slony 2.0, имеется в виду Slony-I версии 2.0.

Подготовка

Первым возникает вопрос: зачем использовать Slony? Дело в том, что в некоторых случаях у проекта имеются преимущества даже по сравнению с потоковой репликацией. Вот некоторые из них:

- Slony реплицирует данные между различными мажорными версиями PostgreSQL;
- допускается реплицировать данные между различными операционными системами с различной аппаратной комплектацией;
- изменения можно записывать в файл и потом пакетно применять при помощи некоторых механизмов. Таким образом, репликацию выполняют, обмениваясь носителями. Это значительно облегчает работу, если сетевое соединение работает

¹ Название Slony происходит от русского слова «слоны» – так выглядит логотип PostgreSQL. Кроме того, Вадим Михеев, один из ключевых разработчиков ядра PostgreSQL в конце 1990-х – начале 2000-х годов, родом из России. – *Прим. ред.*

нерегулярно. Обновления выполняются периодически с удаленных или мобильных устройств;

- допускается выполнять выборочную репликацию;
- можно реплицировать разные наборы таблиц на разные резервные серверы – выполнять развертывание (fan-out). Обратное действие, когда резервный сервер получает таблицы с разных главных серверов, называется *свертыванием* (fan-in, или roll-up);
- Slony позволяет ретранслировать репликацию, так что главный сервер (источник) посылает изменения резервному (подписчику), который (уже в роли провайдера) посылает их другим резервным серверам (также подписчикам).

Slony позволяет создавать кластеры очень сложной конфигурации (к сожалению, тема выходит за рамки данной книги).

Все это звучит очень круто, да так и есть на самом деле. Но будьте осторожны: сложные схемы вредны, когда требуется надежное решение с высокой доступностью данных. Если вам не обойтись без некоторых из описанных возможностей, то используйте Slony. Если необходимости нет или вы не уверены, используйте более простой вариант – потоковую репликацию.

Второй вопрос: какую версию Slony установить? Версия 1.2 используется на многих рабочих системах. В *версии* 2.0 многие характеристики улучшены, она совместима с PostgreSQL 8.3 и более поздними версиями. В Slony 2.0 несколько уменьшено блокирование, она работает более эффективно и гладко. В Slony 2.0.5 решены многие из проблем предыдущих версий, и теперь пользователи отдают предпочтение именно ей¹.

Как это сделать

Как вы могли уже догадаться, настройка Slony сложна. Было несколько попыток ее упростить, но так и не был выработан единый стандарт. В этой книге используется версия Slony1-ctl, которая доступна по адресу <http://pgfoundry.org/projects/slony1-ctl/>.

Преимущества использования Slony1-ctl следующие:

- управление сложными системами репликации с каскадами, несколькими наборами репликации, несколькими главными серверами и тому подобным;

¹ В октябре 2011 вышла версия Slony 2.1.0. На данный момент актуальные версии более старых веток – 1.2.23 и 2.0.7 (см. страницу проекта по адресу <http://slony.info/>). – *Прим. ред.*

- переключение и переключение при отказе (даже с каскадами);
- добавление или удаление объектов (таблиц, последовательностей) с помощью одного простого скрипта;
- добавление и управление узлами и путями между узлами;
- добавление репликационных наборов;
- обновление Slony до следующих версий.

Ключевым моментом является то, что все действия выполняются в два этапа. Первый этап – создание скрипта, второй этап – выполнение скрипта. При этом администратор может изменить скрипт при необходимости или просто остановиться и подумать.

Создаваемые скрипты автономны: для их запуска не нужны компоненты Slony. Многие используют их для создания новых скриптов, которые затем выполняют. Обеспечивается также обработка ошибок.

Процесс полной репликации

В этом рецепте описывается настройка репликации с помощью Slony с одного главного сервера на один резервный. Освоив простую настройку, вы сможете подумать и о более сложных комбинациях. Выполните следующее:

1. Убедитесь, что для каждой реплицируемой таблицы определен первичный ключ или уникальный индекс.
2. Установите Slony на главный и резервный серверы.
3. Создайте учетную запись для выполнения репликации (например, `my_slony_user`) в каждом кластере PostgreSQL. Пользователь, выполняющий репликацию с помощью Slony, должен иметь надежный пароль, шифруемый с помощью md5. Авторизуйтесь на каждом из серверов, участвующих в репликации, и выполните следующую команду от имени пользователя `postgres`:

```
createuser -s -W -E my_slony_user
```

4. Подкорректируйте конфигурационный файл PostgreSQL `pg_hba.conf`. На каждом сервере, участвующем в репликации, настройте авторизацию так, чтобы пользователь `my_slony_user` мог подключаться с любого сервера к базам данных, которые нужно реплицировать.

```
host my_db my_slony_user 192.168.1.111/32 md5
host my_db my_slony_user 192.168.1.112/32 md5
```

5. Сохраните пароль в файле `.pgpass`. Отредактируйте файл `~/.pgpass`, добавив для каждого узла такую строку:

```
192.168.0.1:5432:my_db:my_slony_user:my_slony_pass
```

6. Отредактируйте файл `etc/slony_include.h` в отмеченных местах:

- измените путь к файлам;
- измените имя пользователя Slony.

7. Отредактируйте файл `etc/bases.h`. В нем содержится информация о подключениях для каждой БД, участвующей в репликации. Здесь могут быть разные имена баз данных для одной репликации или одинаковые имена для разных репликаций.

#	CLUSTER	NODE	BASE	HOST	PORT
my_replica	1		my_db	192.168.1.12	5432
my_replica	2		my_db	192.168.1.13	5432

8. Отредактируйте файл `etc/relations.h`. Измените имена таблиц, участвующих в репликации:

#	CLUSTER	SET	MASTER	SLAVE
my_replica	1	1		2

9. Отредактируйте файл `etc/slony.cfg`, если хотите изменить значения по умолчанию.

10. Используйте `slony-ctl`:

```
./01_create_init.sh -c my_replica
./create_struct.sh -c my_replica
./02_exec_init.sh -c my_replica
```

После выполнения последних трех команд БД будут реплицированы и демоны `slon` запущены:

- `01_create_init.sh` создает скрипты `bash` для последующего использования. Можно отредактировать файл `$SLON_TMP/my_replica.sh` и добавить или удалить таблицы, если потребуется. Выборочность репликации осуществляется, таким образом, простым редактированием этих файлов;
- `create_struct.sh` создает дампы всей схемы базы данных на главном сервере и восстанавливает ее на резервном сервере;
- `02_exec_init.sh` запускает созданные скрипты. На этом этапе копируются данные, поэтому он может выполняться долго.

Поддержка репликации

Slony не реплицирует следующие изменения:

- определений таблиц (команды DDL);
- учетных записей и ролей;
- больших объектов (BLOB);
- при использовании команды `TRUNCATE1.....`

¹ Эта возможность уже появилась в версии 2.1.0. – *Прим. ред.*

Итак, после настройки Slony понадобится применять все дальнейшие изменения к реплицируемым таблицам на каждом узле. В составе Slony есть инструмент *slonik*, который позволяет выполнять команду `EXECUTE SCRIPT` и таким образом запустить скрипт с изменениями в одной транзакции на каждом узле.

Без всяких проблем можно создавать новые неуникальные индексы любого типа для таблиц на узле подписчика. Это пригодится, например, при настройке резервных серверов для работы с нагрузкой разных типов. Не забывайте, что нельзя создавать на узле подписчика уникальные индексы, которых нет на главном сервере, иначе могут возникнуть ошибки репликации.

Таблицы на резервных серверах, не участвующие в репликации, можно изменять, чтобы выполнять постобработку присылаемых изменений или создание таблиц в стиле материализованных представлений.

Как это работает

Все изменения помещаются в таблицы логов `sl_log_1` и `sl_log_2` с помощью триггеров в базе данных источника. Slony будет переключаться от одной таблицы к другой, и пока одна таблица является текущей, Slony выполняет сокращение другой таблицы. Это позволяет обойтись без удаления и сборки мусора в таблицах логов. Триггеры также ставятся на таблицы резервных серверов, чтобы предотвратить запись в эти таблицы.

Для каждого узла в кластере создается один процесс *slon*, который пересылает данные и изменения конфигурации с главного сервера на резервный. Демон *slon* читает таблицу логов на источнике (провайдере) и выполняет команды SQL на подписчике (резервном сервере). Пакеты изменений применяются, размер пакета – настраиваемый параметр.

В версии 1.2 Slony все реплицируемые таблицы блокировались при каждом вызове `EXECUTE SCRIPT`, даже если они находились в разных репликационных наборах. В версии 2.0 блокирование таблиц стало менее жестким, хотя это может приводить к недостаточному блокированию и ошибкам.

И еще

Можно управлять Slony с помощью графических оболочек *pgAdmin* и *phpPgAdmin*, но такой способ не всегда годится: управление Slony слишком сложно. На этот счет есть разные мнения.

Смотрите также

Дополнительная информация доступна по адресу <http://slony.info>.

Балансировка нагрузки с помощью pgpool-II 3.0

pgpool-II является программным обеспечением промежуточного слоя (*middleware*), которое может выполнять следующие основные функции:

- управление пулом соединений;
- балансировка нагрузки;
- управление репликацией;
- параллелизация запросов.

Здесь рассматривается только режим балансировки нагрузки, который спроектирован для работы с другими формами репликации, описанными в этой главе. pgpool активно развивается и содержит много новых возможностей для PostgreSQL 9.x. Режим балансировки нагрузки позволяет приложению выполнить одно подключение к pgpool, а затем распределять нагрузку на несколько узлов. В результате существующее приложение можно с минимальными изменениями использовать с pgpool. Другие способы распределения нагрузки на несколько серверов используют программное обеспечение, не связанное с PostgreSQL, и поэтому здесь не обсуждаются.

Подготовка

Прежде чем использовать pgpool, следует разобраться, как работает балансировка нагрузки и будет ли она вам нужна.

В режиме балансировки нагрузки pgpool принимает решения по маршрутизации запросов в зависимости от их типа. При обычном запросе `SELECT` нагрузка распределяется между главным сервером и резервным, если возможно. Другие типы команд SQL направляются только на главный сервер, но никогда не отсылаются на резервный.

Обычные запросы `SELECT` легко балансируются. Обычный запрос (*straightforward*) – это команда `SELECT`, которая производит следующие действия:

- выполняется как запрос верхнего уровня, запросы внутри функций не реплицируются;

- выполняется не как часть явной или расширенной транзакции;
- не использует блокирование на уровне строк `SELECT ... FOR SHARE/UPDATE;`;
- не применяет функции `nextval()` или `setval()`;
- не вызывает функции класса `volatile`;
- не обращается к таблицам системных каталогов.
- Более подробную информацию о pgpool можно найти по адресу <http://pgfoundry.org/projects/pgpool/>.
- pgpool имеет дополнительные возможности:
- резервные серверы можно добавлять без перезапуска pgpool;
- автоматическое переключение при отказе;
- кэширование запросов во всех режимах.

Как это сделать

При использовании pgpool с потоковой репликацией логов должен быть включен режим горячего резервирования.

Установите pgpool, затем направьте приложения на сервер pgpool, который будет отвечать им как сервер PostgreSQL. Затем pgpool подключается к главному и резервным серверам. Можно отредактировать файл `pool_hba.conf` для управления доступом, как это делается с файлом `pg_hba.conf` для сервера PostgreSQL.

Вам понадобятся следующие опции в файле `pgpool.conf.sample-stream`:

```
replication_mode = false
Master_slave = true
replicate_select = false
load_balance_mode = true
```

Они включают режим балансировки нагрузки. Кроме того, pgpool поддерживает следующие директивы:

- `delay_threshold` определяет приемлемую задержку репликации на резервном сервере по сравнению с главным сервером, измеряемую в байтах. Если задержка репликации превышает это значение, то балансировка нагрузки отключается, и все запросы отправляются на главный сервер. Если вам это не нужно, придайте параметру нулевое значение;
- `log_standby_delay` позволяет регистрировать задержку, чтобы изучить ее изменения впоследствии. При значении `always` время фиксируется при каждой проверке состояния. При значении `if_over_threshold` величина задержки сравнивается с параметром `delay_threshold`, и фиксируется при превышении его;

- `health_check_period` определяет промежуток между измерениями задержки. Параметр `health_check_user` назначает пользователя, от имени которого проводится проверка состояния.

Есть дополнительные параметры для автоматического переключения при отказе. В файле `pgpool.conf.sample-stream` вы найдете более подробную информацию об этой настройке и других параметрах.

Можно управлять `pgpool` из командной строки примерно следующим образом:

```
pgpool -f pgpool.conf -a pool_hba.conf
pgpool -m fast stop
pgpool -f pgpool.conf -a pool_hba.conf reload
```

Величину задержки репликации можно посмотреть интерактивно, выполнив следующее:

```
SHOW POOL_STATUS;
```

Эта команда перехватывается `pgpool` и возвращает информацию о состоянии. Другие команды `SHOW` выдают какую-либо другую информацию.

Как это работает

`pgpool` разбирает команды SQL, чтобы идентифицировать их тип и содержание. Разбор должен быть достаточно подробным, чтобы понять, содержатся ли в запросах вызовы функций класса `volatile`. `pgpool` ничего не знает о функциях, определяемых пользователем, но вы можете включать функции в черный или белый список с помощью параметров `white_function_list` (внести в белый список) и `black_function_list` (внести в черный список).

Проверка состояния производится в отдельном процессе `pgpool`, который не блокирует выполнение SQL.

И еще

`pgpool-II` является программным обеспечением промежуточного слоя, которое может управлять пулом соединений, выполнять балансировку нагрузки и параллелизацию запросов, управлять репликацией.

Некоторые из этих функций не рассматривались, поэтому следует сказать кое-что о них здесь. Управление пулом соединений может выполняться с помощью `pgbouncer`. Управление репликацией, вероятно, лучше осуществлять в одной из форм, которые описаны в этой книге.

Параллелизация запросов, к сожалению, на порядок лучше выполняется такими популярными продуктами, как Greenplum. Его бесплатная версия прекрасно работает и широко применяется многими пользователями PostgreSQL (посетите страницу <http://www.greenplum.com/products/single-node/>).

Минорное обновление

Минорные обновления версий выпускаются регулярно всеми разработчиками программ; то же самое относится и к PostgreSQL. Когда происходит минорный выпуск, увеличивается последнее число в номере версии – обычно на единицу.

Итак, первый мажорный выпуск версии 9.0 – 9.0.0. Первый пакет исправлений – 9.0.1, затем 9.0.2 и т. д.

В этом разделе обсуждается переход с одной минорной версии на другую минорную версию.

Подготовка

Во-первых, раздобудьте новую версию, скачав либо исходный код, либо двоичный код.

Как это сделать

В большинстве случаев обновление до следующей минорной версии PostgreSQL тривиально. Форматы файлов данных, индексов и логов транзакций (WAL) сохраняются. Измениться может только формат некоторых временных файлов.

Процесс обновления происходит следующим образом:

1. Прочитайте замечания к выпуску, чтобы узнать, нужно ли выполнить какие-то особые действия.
2. Если вы пользуетесь профессиональной технической поддержкой, поговорите с теми, кто ее предоставляет, чтобы выяснить, нужны ли какие-либо предварительные проверки кроме тех, что указаны в инструкции по обновлению. Проверьте, совместима ли новая версия с вашим оборудованием, операционной системой и конкретной ее версией, установленной на вашем компьютере.
3. Выполните все проверки. Например, если изменился формат файлов WAL, вам, возможно, придется перенастроить

файловую репликацию после обновления. Вы должны будете просканировать таблицы, воссоздать индексы или сделать нечто подобное. Не для каждого обновления потребуются такие действия, однако упускать их из виду не следует — они очень важны.

4. Если вы применяете репликацию, проверьте результат обновления, перезапустив один из резервных серверов.
5. Выполните инструкции, относящиеся к используемому вами дистрибутиву операционной системы и способу установки программ. Эти инструкции могут существенно различаться.
6. Запустите сервер баз данных, выделенный для тестирования, выполните все специальные постустановочные действия и проверьте, все ли работает должным образом.
7. Выполните шаги 4–6 на других резервных серверах.
8. Выполните шаги 4–6 на главном сервере.

Как это работает

Минорные обновления в основном касаются исполняемого кода, поэтому процедура должна свестись к замене двоичных файлов новыми версиями и перезапуску. Но все же будьте внимательны.

Мажорные обновления на месте

В PostgreSQL 9.0 появилась новая утилита `pg_upgrade` в составе модулей `contrib`. Она позволяет переходить от одной версии PostgreSQL к другой мажорной версии, например от 8.4 к 9.0 или от 9.0 к 9.1.

Обновление на месте означает обновление баз данных без создания нового сервера. Звучит хорошо, но следует учитывать потенциальные недостатки:

- на время обновления сервер баз данных должен быть выключен;
- в системе должно быть достаточно места для двух копий сервера баз данных, старой и новой. Если это не так, придется задействовать утилиту `pg_upgrade` в варианте со ссылками или воспользоваться рецептом обновления с помощью репликации. Если применяются ссылки, то откат обновления с помощью утилиты `pg_downgrade` невозможен. Можно только выполнить восстановление из резервной копии, и это означает длительное выключение сервера;

- если вы сначала сделаете копию баз данных, а потом выполните обновление, то потраченное на это время пропорционально размеру баз данных;
- утилита `pg_upgrade` поддерживает только PostgreSQL версии 8.3 и последующие;
- утилита `pg_upgrade` не проверяет все ваши дополнительные модули, и вам придется самостоятельно провести тестирование их совместимости с новой версией перед тем, как выполнить обновление (извините, пока нет легких способов сделать это!).

Подготовка

Определите размер своих баз данных (найдите соответствующий рецепт выше). Если БД велики или данные должны быть постоянно доступны, подумайте об обновлении с использованием репликации. Изучите нижеследующий материал.

Как это сделать

1. Прочитайте замечания к новой версии. Обратите особое внимание на раздел о проблемах несовместимости.
2. Создайте тестовый сервер со старой версией программного обеспечения. Обновите до новой версии, чтобы проверить, нет ли конфликтов зависимости программ. Протестируйте свое приложение. Найдите и проверьте каждый используемый вами дополнительный модуль PostgreSQL, убедитесь, что они работают корректно с новой версией PostgreSQL.
3. Сделайте резервную копию сервера. Готовьтесь к худшему и надейтесь на лучшее.
4. Напишите тестовый скрипт, который должен подтвердить, что обновление прошло успешно. Используйте команды, которые выполняются успешно только на новой версии. Пример:

```
SELECT (string_to_array(version(), ' ')[2] = '9.0.1');
```
5. Заранее подумайте, кого звать на помощь в случае неприятностей и как именно выполнить восстановление из резервной копии, которую вы только что сделали.
6. Установите новые версии всех нужных программ на рабочий сервер и создайте новый сервер баз данных.
7. Не снижайте уровень безопасности на время обновления. Сотрудники отдела безопасности придут в ярость, если узнают об этом. Не потеряйте работу!

8. Выключите серверы баз данных.
9. Запустите `pg_upgrade`, а потом – все скрипты, необходимые после обновления. Не забудьте проверить, какие именно действия нужно выполнить.
10. Запустите новый сервер баз данных и сразу же выполните команду `ANALYZE` для всего сервера.
11. Прогоните тесты, чтобы проверить, прошло ли обновление успешно или нужно отходить на прежние позиции.
12. Если все в порядке, включите доступ к базам данных. Перезапустите приложения.
13. Не удаляйте каталог прежнего сервера, если использовался метод со ссылками. Прежний каталог данных все еще содержит данные нового сервера. Это сложная ситуация, не допускайте ошибок!

Как это работает

Работа утилиты `pg_upgrade` проста, так как формат блока данных не изменяется между версиями. Но так бывает не всегда. Обновление через репликацию тоже может не сработать между мажорными версиями, хотя с последними семью версиями PostgreSQL все проходило успешно.

Утилита `pg_upgrade` создает сначала новый набор таблиц каталогов базы данных, затем воссоздает старые объекты в новых таблицах. Обычно никаких проблем не возникает.

И еще

Обновление через несколько мажорных версий также возможно. Однако это делается последовательно: сначала до следующей версии, потом снова до следующей. Конечно, вся процедура усложняется, но последовательность действий та же.

Тщательно подготавливайте тесты, которые должны сигнализировать об успешном обновлении.

Мажорные онлайн-обновления с помощью инструментов репликации

Трудно выполнять обновление между мажорными версиями – делайте это только при необходимости и правильно выбирайте время.

Инструменты репликации можно использовать для минимизации времени недоступности сервера, поэтому в настоящем рецепте речь пойдет об *онлайн-обновлении* (online upgrade).

Как это сделать

Выполните следующие пункты, начав подготовку как минимум за месяц, чтобы все протестировать и чтобы каждый понял, как будет проходить процесс:

1. Установите новую версию программного обеспечения на новую систему.
2. Сделайте автономную резервную копию главной системы и создайте тестовый сервер. Тщательно протестируйте приложения с новой версией сервера на тестовой системе. Если все работает как нужно, переходите к следующему шагу.
3. Создайте пул подключений для главной базы данных (если его еще нет).
4. Настройте репликацию с использованием Londiste или Slony, как это описано в предыдущих рецептах.
5. Снова протестируйте приложения с новой версией сервера на актуальных данных.
6. Приготовьте конфигурацию нового пула соединений, указывающую на новую систему.
7. Остановите пул соединений.
8. Переключитесь на новую систему.
9. Переключите пул соединений на новую систему и перезапустите.

Как это работает

Slony и Londiste работают со многими версиями PostgreSQL, и вы можете быть уверены, что с этой стороны все будет хорошо. Предыдущий рецепт предусматривает нулевые потери данных, так как используется процесс чистого переключения. Нет необходимости надолго отключать сервер во время обновления, и снижен риск по сравнению с обновлением на месте. Обновление сервера – также удачный момент для обновления или изменения конфигурации оборудования. Кроме того, в это время можно изменить кодировку на сервере.

Администрирование PostgreSQL 9. Книга рецептов

Получите ответы на все часто возникающие вопросы по PostgreSQL. Заставьте ваши базы данных работать более эффективно, сделайте их более управляемыми и безопасными.

Авторы книги, Саймон Ригс и Ханну Кросинг, обладающие уникальным опытом не только работы с PostgreSQL, но и разработки самых критически важных его частей, делятся с вами своим опытом, чтобы помочь вам работать с PostgreSQL эффективно. Эта книга представляет собой большой набор практических решений по наиболее острым и часто встречающимся у администраторов баз данных и разработчиков проблемам – от вопросов разработки до поддержки.

Эта книга...

- помогает освоить все возможности PostgreSQL для высокой производительности и доступности ваших сервисов;
- описывает функции самых последних версий PostgreSQL;
- полезна для администраторов баз данных, обладающих опытом работы с другими СУБД;
- содержит продвинутые технические рецепты и скрипты для опытных пользователей;
- предлагает лучшие рецепты для планирования и проектирования баз данных, предназначенных для работы в режиме 24/7;
- рассказывает, как и когда выполнять VACUUM для ваших баз данных;
- сравнивает различные техники выполнения быстрого резервного копирования и восстановления баз данных;
- предлагает краткое и понятное руководство по репликации и достижению режима высокой доступности ваших баз данных.



Внутри книги вы найдете:

- понятный и доступный для освоения формат;
- выборку наиболее важных задач и проблем;
- тщательно отобранные и организованные инструкции для эффективного решения возникающих проблем;
- ясные объяснения ваших действий;
- указания, как применять описанные решения к другим ситуациям.



Internet-магазин www.dmk-press.ru
Книга-почтой: orders@aliens-kniga.ru
Оптовая продажа: «Альянс-книга»
(499)725-5409, books@aliens-kniga.ru

ISBN 978-5-94074-750-5



9 785940 747505 >