O'REILLY®

ЭФФЕКТИВНЫЙ Spark

Масштабирование и оптимизация

Холден Карау
Рейчел Уоррен

ПИТЕР®

High Performance Spark

Best Practices for Scaling and Optimizing Apache Spark

Holden Karau and Rachel Warren

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

ЭФФЕКТИВНЫЙ Spark

Масштабирование и оптимизация

Холден Карау
Рейчел Уоррен

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

2018

ББК 32.972.233.02
УДК 004.62
К21

Холден Карау, Рейчел Уоррен
К21 Эффективный Spark. Масштабирование и оптимизация. — СПб.: Питер, 2018. — 352 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-0705-6

Если у вас уже есть положительный опыт использования Spark для решения небольших задач, но вы по-прежнему ломаете голову — где та самая непревзойденная производительность Spark, позволяющая перемалывать колоссальные объемы данных, — то эта книга для вас. Она расскажет, как эффективно использовать Spark для укрощения больших данных и вырасти из новичка в специалиста. Идеально подходит для программистов, инженеров по работе с данными, системных администраторов, обслуживающих крупномасштабные приложения.

16+ (Для детей старше 6 лет. В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.972.233.02
УДК 004.62

ISBN 9781491943205 англ.

ISBN 978-5-4461-0705-6

Краткое содержание

Оглавление

Предисловие

Мы написали эту книгу для дата-инженеров (data engineers) и ученых по данным (data scientists), которым хотелось бы извлечь максимальную пользу из фреймворка Spark. Если вы работаете со Spark и вложили в его освоение немало усилий, но погрязли в ошибках памяти и загадочных, перемежающихся сбоях, наша книга для вас. Если вы применяли Spark в какой-либо исследовательской деятельности или экспериментировали с ним, не используя для конкретных производственных задач, и запустить фреймворк в работу мешает недостаточная уверенность в уровне его освоения, то это издание может помочь. Если вы энтузиаст Spark, но не сумели добиться от него ожидаемого повышения производительности, то мы надеемся, что вы найдете здесь ответы на свои вопросы. Книга предназначена для тех, кто уже пользовался Spark, и может оказаться не вполне понятной тем, кто до сих пор мало работал с этим фреймворком или с распределенными вычислениями либо вообще не имел с ними дела. В разделе «Вспомогательная литература и материалы» чуть ниже можно найти рекомендации по книгам для начинающих.

Мы предполагаем, что материал окажется наиболее полезен тем, кому важна оптимизация часто повторяемых запросов в эксплуатационной среде, а не тем, кто занимается в основном научной работой. Хотя написание высокопроизводительных запросов важнее всего для дата-инженеров, создание запросов с помощью Spark (в отличие от других фреймворков) требует глубоких знаний данных, присущих скорее ученым по данным. Следовательно, издание будет полезнее для тех дата-инженеров, которые в вопросах производительности менее склонны рассуждать критически относительно статистической природы, распределения и форматирования данных. Мы надеемся, что книга поможет вам более критично относиться к своим данным при вводе конвейеров (pipelines) в промышленную эксплуатацию. Хотелось бы, чтобы вы задавали себе такие вопросы: «Как распределены мои данные?», «Асимметричны ли они?», «Каков диапазон значений столбца?» и «Как группируется данная величина?», и в зависимости от ответов на них формировали логику своих Spark-запросов.

Впрочем, даже тем ученым по данным, кто использует этот фреймворк в основном для исследовательской деятельности, книга способна дать представление о том, как писать производительные запросы на Spark. Так что по мере неизбежного роста масштаба исследований вы сможете с первого же раза добиться требуемых результатов. Мы надеемся указать верную дорогу специалистам по науке о данных (даже тем, кто уже привык к распределенному представлению данных и умеет критически оценивать свои программы), помочь им исследовать свои данные более полно и быстро, а также эффективно взаимодействовать с другими участниками ввода программ в эксплуатацию.

Независимо от вашей должности есть вероятность, что объем данных, с которыми приходится иметь дело, растет очень быстро. Использовавшиеся изначально программные решения могут потребовать масштабирования, а старые методы — обновления, чтобы можно было работать над новыми проектами. Надеемся, эта книга поможет вам применять фреймворк Apache Spark, чтобы решать новые задачи проще, а старые — эффективнее.

Примечания к первому изданию

Вы читаете первое издание книги *High Performance Spark* («Эффективный Spark»), и мы благодарны за это! В случае обнаружения ошибок, опечаток или появления идей, способных улучшить книгу, пожалуйста, напишите нам по адресу high-performance-spark@googlegroups.com. Если при этом вы хотите, чтобы мы упомянули вас в разделе благодарностей будущих изданий этой книги, пожалуйста, укажите имя, под которым вы хотели бы фигурировать.

Вспомогательная литература и материалы

Великолепное вводное руководство[1] для исследователей данных и разработчиков, еще не знакомых со Spark, — *Learning Spark* Матея Захариа (Matei Zaharia), Холдена Карау (Holden Karau), Энди Конвински (Andy Konwinski), Патрика Венделла (Patrick Wendell)[2] (http://shop.oreilly.com/product/0636920028512.do). Замечательная книга для исследователей данных, интересующихся Spark, — *Advanced Analytics with Spark* Сэнди Риза (Sandy Ryza), Ури Лезерсона (Uri Laserson), Шона Оуэна (Sean Owen) и Джоша Уиллса (Josh Wills)[3] (http://shop.oreilly.com/product/0636920035091.do).

---

1 Хотя, возможно, в данном случае мы пристрастны.

2 Изучаем Spark. Молниеносный анализ данных / Холден Карау, Энди Конвински, Патрик Венделл, Матей Захария. — М.: ДМК Пресс, 2015. — 304 с.

3 Spark для профессионалов: современные паттерны обработки больших данных / С. Риза, У. Лезерсон, Ш. Оуэн, Д. Уиллс. — СПб.: Питер, 2017. — 272 с.

Для тех, кого больше интересует потоковая обработка, может оказаться полезной книга *Learning Spark Streaming*, написанная Франсуа Гарильо (Francois Garillot) (http://shop.oreilly.com/product/0636920047568.do), планируемая к выходу в ближайшее время.

Помимо книг, доступно множество обучающих материалов по фреймворку Spark. Для тех, кто предпочитает видео, Пако Натан (Paco Nathan) создал великолепную серию видеоуроков для начинающих на сайте O'Reilly (http://shop.oreilly.com/product/0636920036807.do). Databricks (https://databricks.com/training) и Cloudera (https://www.cloudera.com/more/training/courses/spark-training.html), а также другие поставщики Hadoop/Spark предлагают обучать работе с фреймворком на платной основе. На странице документации по языку Spark от корпорации Apache (http://spark.apache.org/documentation.html) выложены записи учебных семинаров по Spark и много других замечательных материалов.

Если вы еще не работали с языком программирования Scala, то в главе 1 мы постараемся убедить вас применять его. А если хотели бы его изучить, то рекомендуем отличный вводный курс *Programming Scala, 2nd Edition* Дэна Уомплера (Dean Wampler) и Алекса Пэйна (Alex Payne)[1] (http://shop.oreilly.com/product/0636920033073.do).

Условные обозначения

В данной книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины.

`Моноширинный шрифт`

Применяется для листингов программ, а также внутри абзацев, чтобы обратиться к элементам программы, таким как переменные или имена функций, базы данных, переменные окружения, операторы и ключевые слова. Им также выделены имена и расширения файлов.

`Полужирный моноширинный шрифт`

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

`Курсивный моноширинный шрифт`

Показывает текст, который должен быть заменен значениями, введенными пользователем, или значениями, определяемыми контекстом.

[1] Хотя не помешает отметить, что некоторые из предлагаемых в этой книге практик не являются общепринятыми для Scala.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты.

Этот рисунок указывает на совет или предложение.

Этот рисунок указывает на общее замечание.

Этот рисунок указывает на предупреждение.

Примеры, которые предваряет этот рисунок, в значительной степени зависят от внутреннего устройства фреймворка Apache Spark и, вероятно, перестанут работать в следующих же младших версиях выпусков Apache Spark. Мы вас предупредили, но легко поймем, если вы не станете уделять этому особого внимания, как, впрочем, и мы.

Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и т. п.) доступны для скачивания из GitHub-репозитория к данной книге (https://github.com/high-performance-spark/high-performance-spark-examples), а часть тестового кода находится в GitHub-репозитории «Тестовая база Spark» (https://github.com/holdenk/spark-testing-base) и репозитории Spark Validator (https://github.com/holdenk/spark-validator). Структурированные примеры потокового машинного обучения, обычно относящиеся к категории, обозначенной изображением скорпиона (обсуждавшимся в разделе «Условные обозначения» выше), доступны на https://github.com/holdenk/spark-structured-streaming-ml.

Эта книга создана, чтобы помочь вам в работе. В целом приведенные здесь примеры кода вы можете применять в своих программах и документации. Обращаться к нам за разрешением нет необходимости, если только вы не копируете значительную часть кода. Так, написание программы с использованием нескольких фрагментов кода из книги не требует отдельного разрешения. Для продажи или распространения компакт-диска с примерами из книг O'Reilly оно, конечно, необходимо. Ответ на вопрос путем цитирования этой книги и примеров кода не требует разрешения. Код также доступен под лицензией Apache 2. Включение значительного количества примеров кода из этого издания в документацию к вашему продукту может потребовать разрешения.

Мы ценим, хотя и не требуем, ссылки на первоисточник. Такая ссылка включает название, автора, издательство и ISBN. Например: «*Карау Холден, Уоррен Рейчел*. Эффективный Spark. Масштабирование и оптимизация. — О'Рейли, 2018. — 978-1-491-94320-5».

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

Как связаться с авторами

Замечания и предложения по книге вы можете направлять по электронной почте: high-performance-spark@googlegroups.com. Чтобы почитать всякие глупости, которые мы пишем, иногда и насчет Spark, можете подписаться на нас в Twitter:

- Холден — http://twitter.com/holdenkarau;
- Рейчел — https://twitter.com/warre_n_peace.

Благодарности

Мы хотели бы поблагодарить всех, кто помогал нам замечаниями и предложениями по черновикам книги. Отдельные благодарности Ане Байда (Anya Bida), Джейкобу Одерски (Jakob Odersky) и Кэтрин Кирнан (Katharine Kearnan) за отзывы о предварительных набросках и рисунках. Мы хотели бы выразить благодарность Махмуду Ханафи (Mahmoud Hanafy) за обзор и усовершенствование кода примеров, равно как и черновиков книги. Мы хотели бы также поблагодарить Майкла Армбраста (Michael Armbrust) за отзывы и замечания по предварительной версии главы, посвященной SQL. Одним из наиболее активных читателей первоначальной версии издания был Джастин Пифони (Justin Pihony), предложивший исправления в книге во всех аспектах (язык, форматирование и т. д.).

Спасибо всем читателям предварительного издания книги, приславшим свои замечания о различных ошибках. Отдельная благодарность Канак Кшетри (Kanak Kshetri) и Рубену Беренгуэлю (Rubén Berenguel).

Наконец, спасибо нашим уважаемым работодателям за понимание. Особенно Лоуренсу Спраклину (Lawrence Spracklen), который настоял на том, чтобы мы его тут упомянули :p.

1 Введение в эффективный Spark

В данной главе приведен обзор всего, что вы, надеемся, сможете узнать из нашей книги. Мы постараемся убедить вас выучить язык программирования Scala. Спокойно переходите сразу к главе 2, если знаете, что вам нужно, и уже используете Scala (или твердо решили писать на другом языке программирования).

Что такое Spark и почему производительность так важна

Фреймворк Apache Spark — высокопроизводительная универсальная распределенная система вычислений, самая активная часть проекта с открытым исходным кодом Apache более чем с 1000 участников[1]. Spark обеспечивает возможность обработки больших массивов данных, помимо тех, что могут уместиться на одной машине, с помощью высокоуровневого, относительно простого в использовании API. Spark — одна из самых быстрых систем среди аналогов, его архитектура и интерфейс уникальны. Это единственная система, которая позволяет описывать логику преобразований данных и алгоритмов машинного обучения так, чтобы не зависеть от системы, но сохранить возможность параллельного выполнения. Поэтому данный фреймворк зачастую используют для написания вычислений, которые будут работать быстро в распределенных системах хранения различных видов и размеров.

Однако, несмотря на множество преимуществ Spark и шумиху вокруг него, простейшие реализации многих распространенных стандартных операций науки о данных на Spark будут работать медленнее и менее надежно, чем оптимальные версии. А в силу того, что интересующие нас вычисления касаются обработки данных в огромном масштабе, выгоды от тонкой настройки производительности

[1] С сайта http://spark.apache.org/

кода могут быть колоссальными. Производительность — это не только быстрая работа, в подобном масштабе зачастую речь идет о том, чтобы вообще работать хоть как-то. Можно создать запрос Spark, который не будет выполняться при гигабайтных объемах данных, но после рефакторинга и внесения поправок в структуру конкретных данных и требования кластера прекрасно заработает в той же системе с терабайтами данных. Мы встречались в нашей практике написания промышленного кода Spark с тем, что одни и те же задачи на тех же кластерах работали в сотни раз быстрее после описанных здесь оптимизаций. Говоря языком обработки данных, время — деньги, и мы надеемся, эта книга окупит себя снижением стоимости информационной инфраструктуры и экономией времени разработчиков.

Далеко не все из этих методик применимы для каждого сценария использования. Spark является чрезвычайно гибким и более высокоуровневым, чем другие сравнимые по возможностям вычислительные фреймворки. Именно это позволяет извлечь колоссальную выгоду просто из более точной подгонки под форму и структуру данных. Некоторые из методов будут хорошо работать с определенными объемами данных или даже при определенных распределениях ключей, но не все. Простейший пример: во многих задачах использование функции `groupByKey` в Spark может с легкостью привести к ужасающим исключительным ситуациям из-за нехватки памяти. Но в отношении данных с незначительным дублированием эта операция будет работать столь же быстро, как и ее альтернативы, с которыми мы вас познакомим. Глубокое понимание конкретного сценария применения и системы, а также взаимодействие с ними фреймворка Spark совершенно необходимы для решения с его помощью наиболее сложных задач науки о данных.

Что даст эта книга

Мы надеемся, что, после того как вы прочтете книгу, ваши Spark-запросы станут быстрее, смогут обрабатывать бо́льшие объемы данных и будут потреблять меньше ресурсов. Издание охватывает широкий диапазон утилит и сценариев. Возможно, вы найдете здесь методики, неприменимые к вашим текущим задачам, но способные пригодиться для решения задач будущих, а также расширяющие ваше понимание фреймворка Spark. Главы этой книги содержат достаточно материала, чтобы ее можно было использовать как справочник. Однако структура издания неслучайна: читая главы по порядку, вы сможете в целом понять суть фреймворка Apache Spark и узнаете, как заставить его работать по-настоящему.

В равной степени важно отметить то, чего эта книга, вероятнее всего, не даст. Она не задумывалась как введение в Spark или Scala; для начинающих есть несколько других книг и серий видеоуроков. Возможно, мы несколько пристрастны в данном вопросе, но книга *Learning Spark*, как и великолепная серия видеоуроков для начинающих Пако Натана (Paco Nathan) (http://shop.oreilly.com/product/0636920036807.do), — отличные варианты для начинающих изучать Spark.

Хотя мы сосредоточимся на производительности, книга не предназначена для обслуживающего персонала, поэтому такие вопросы, как настройка кластера и мультиарендность, не рассматривались. Мы предполагаем, что вы уже знаете, как использовать Spark в вашей системе, вследствие чего не предложим особой помощи в вопросе принятия высокоуровневых архитектурных решений. Планируются к выходу книги других авторов, посвященные обслуживанию фреймворка Spark, которые должны уже выйти к моменту чтения вами нашей книги. Если вы как раз занимаетесь обслуживанием или в вашей организации нет выделенного персонала для данной цели, надеемся, что эти книги помогут.

Версии Spark

Spark придерживается семантики контроля версий вида [старшая версия].[младшая версия].[обновление] с сохранением неизменности API для общедоступных, не экспериментальных и не предназначенных для разработчиков API в пределах старших и младших версий. Многие экспериментальные компоненты, включая `Datasets` — новый структурированный и сильно типизированный слой абстракции Spark для SQL, — самые интересные с точки зрения производительности. Фреймворк также старается добиться бинарной совместимости API версий с помощью MiMa[1]; поэтому при использовании стабильного API для выполнения задания в новой версии Spark повторная компиляция обычно требуется разве что при смене старшей версии.

При написании данной книги использовался API Spark 2.0.1, но большая часть кода будет работать и с более ранними версиями фреймворка. Мы старались отмечать те места, в которых это не так.

Почему Scala

В этой книге мы сосредоточимся на API Scala фреймворка Spark и предполагаем, что вы немного знаете этот язык. Частично такое решение принято в целях экономии времени и места; мы верим, что читатели, желающие использовать Spark с другим языком программирования, смогут понять примененные в этой книге концепции, не прибегая к переводу примеров на Java и Python. Но важнее то, что авторы верят: «настоящую» эффективную разработку на Spark легче всего проводить на языке программирования Scala.

Для ясности: описанные причины относятся исключительно к использованию Spark со Scala, существует множество более общих доводов за (и против) приложений на Scala в других контекстах.

[1] MiMa — программа управления миграцией (Migration Manager) для языка программирования Scala, отслеживающая бинарные несовместимости разных версий.

Чтобы стать специалистом по Spark, все равно нужно хоть немного разбираться в Scala

Хотя языки программирования Python и Java более распространены, для желающих поближе познакомиться с разработкой на Spark изучение Scala вполне оправдывает себя. В документации фреймворка могут быть неточности. В то же время удобочитаемость базы кода очень высока. Вероятно, глубокое понимание кодовой базы Spark важнее для продвинутого пользователя Spark, чем в случае других фреймворков. А поскольку Spark написан на Scala, будет непросто взаимодействовать с его исходным кодом без способности по крайней мере читать код на языке Scala. Более того, методы класса *Resilient Distributed Datasets* (RDD, отказоустойчивые распределенные наборы данных) очень напоминают методы API коллекций языка Scala. Спецификации функций RDD, например `map`, `filter`, `flatMap`, `reduce` и `fold`, практически идентичны их эквивалентам языка Scala[1]. По существу, Spark — функциональный фреймворк, в значительной степени основанный на таких концепциях, как неизменяемость и лямбда-выражения, поэтому некоторые знания функционального программирования могут значительно облегчить использование API Spark.

API фреймворка Spark для языка Scala легче использовать, чем API языка Java

Разобравшись со Scala, вы быстро обнаружите, что писать на нем приложения Spark намного удобнее, чем на языке Java. Во-первых, написание таких приложений на Scala требует намного меньшего количества кода, чем на Java, поскольку Spark в значительной степени полагается на встроенные определения функций и лямбда-выражения, которые гораздо более естественно поддерживаются в языке Scala (особенно справедливо это было до выхода Java 8). Во-вторых, командная оболочка Spark — мощный инструмент отладки и разработки, а она доступна только в языках, для которых существуют REPL (Scala, Python и R).

Язык Scala производительнее, чем Python

Может показаться заманчивым создавать приложения Spark на языке Python, поскольку он прост в изучении, писать код на нем легко, это интерпретируемый язык программирования. Кроме того, для него существует обширный набор инструментария для науки о данных. Однако написанный на языке Python код Spark часто работает медленнее, чем эквивалентный код, написанный для JVM, поскольку Scala — язык программирования со статической типизацией, а затраты на взаимодействие с JVM (между Python и Scala) могут быть очень высоки. Наконец, обычно компоненты Spark сначала пишутся на Scala, а только позднее

[1] Хотя, как мы обнаружим в данной книге, их влияние на производительность и семантика вычислений сильно отличаются.

транслируются на Python, так что для использования наиболее передовых возможностей Spark необходима JVM; в частности, особенно отстает поддержка MLlib и Spark Streaming.

Почему не Scala

Существует несколько причин и для того, чтобы вести разработку приложений Spark на других языках программирования. Одна из наиболее важных — предпочтения разработчика или команды. Существующий код, как внутренний, так и библиотечный, может также оказаться серьезным доводом в пользу применения другого языка программирования. Python — один из лучше всего поддерживаемых языков на сегодняшний день. Хотя код на языке Java может быть немного громоздким и иногда слегка отстает в смысле API, производительность при написании на другом языке JVM страдает совсем незначительно (в основном за счет преобразования объектов)[1].

Хотя для итогового издания этой книги все примеры представлены на Scala, мы постараемся перенести многие образцы со Scala на Java и Python в тех случаях, когда различия в реализации существенны. Со временем они будут выложены в нашем GitHub-репозитории (https://github.com/high-performance-spark/high-performance-spark-examples). Если вы хотели бы переноса конкретного примера, то, пожалуйста, напишите нам сообщение по электронной почте или зарегистрируйте соответствующую проблему в репозитории GitHub.

Spark SQL значительно уменьшает различия в производительности при использовании не-JVM-языков. В главе 7 мы рассмотрим варианты эффективной работы со Spark на подобных языках, включая поддерживаемые Spark языки Python и R. Кроме того, в этом разделе приведены рекомендации по применению FORTRAN, C и кода, ориентированного на конкретные GPU, в целях дальнейшего повышения производительности. Даже при разработке большей части Spark-приложения на языке Scala мы не обязаны все делать только на нем, поскольку специализированные библиотеки на других языках программирования вполне могут окупить накладные расходы на выход за пределы JVM.

Изучение Scala

Если нам все же удалось убедить вас использовать Scala, то для его изучения есть несколько замечательных возможностей. Spark 1.6 основан на Scala 2.10 и кросскомпилирован с помощью Scala 2.11, а Spark 2.0 основан на Scala 2.11 и, возможно, кросскомпилирован с использованием Scala 2.10; кроме того, он, вероятно,

[1] Конечно, когда речь идет о производительности, из каждого правила есть исключения. Производительность преобразования mapPartitions в Spark 1.6 и более ранних версиях весьма ограничена, о чем мы поговорим в разделе «Выполнение преобразований “итератор — итератор” с помощью функции mapPartitions» на с. 121.

подвергнется данной операции с применением 2.12 в будущем. В зависимости от того, насколько нам удалось убедить вас изучить Scala, и от ваших возможностей, существует множество различных вариантов, начиная с книг и массовых открытых курсов дистанционного обучения (massive open online course, MOOC) и заканчивая курсами профессионального обучения.

Что касается книг, отлично подойдет *Programming Scala, 2nd Edition* (http://shop.oreilly.com/product/0636920033073.do), хотя многие из приведенных в ней системных ссылок акторов при работе в Spark не годятся. Список книг по Scala приведен и на сайте, посвященном Scala (http://www.scala-lang.org/documentation/learn.html).

Помимо книг, описывающих Scala, существуют онлайн-курсы для изучающих этот язык. На портале Coursera размещен курс Functional Programming Principles in Scala («Принципы функционального программирования на Scala») (https://www.coursera.org/learn/progfun1), преподаваемый его создателем Мартином Одерски (Martin Ordersky), а на портале edX — курс Introduction to Functional Programming («Введение в функциональное программирование») (https://www.edx.org/course/introduction-functional-programming-delftx-fp101x-0). Кроме того, несколько различных компаний тоже предлагают видеокурсы по языку Scala, но мы лично не слушали ни один из них, так что рекомендовать их не можем.

Для предпочитающих более интерактивный подход множество разных компаний, включая Lightbend (бывшая Typesafe), предлагают курсы профессионального обучения (https://www.lightbend.com/services/training). Хотя мы сами не пользовались этими курсами, они получили немало положительных отзывов и известны желанием помочь командам инженеров или группе лиц войти в курс разработки на Scala именно для работы со Spark.

Резюме

Хотя максимальной производительности Spark можно добиться только при условии хорошего понимания языка Scala, работа на Spark не требует знания этого языка. Если для ваших задач лучше подходят другие языки программирования или инструменты, то обратитесь к главе 7, где описаны методики работы с другими языками. Эта книга ориентирована на тех разработчиков, которые уже владеют основами Spark, и мы благодарны вам за то, что углубить свои знания вы решили с помощью нашего издания. В следующей главе мы представим основы архитектуры Spark и базовые принципы вычислений, необходимые для эффективного использования этого фреймворка.

2 Как работает Spark

Эта глава познакомит вас с общей архитектурой фреймворка Spark, а также его местом в экосистеме больших данных. Он часто рассматривается как альтернатива Apache MapReduce, поскольку Spark можно использовать для распределенной обработки данных с помощью Hadoop[1]. Как мы увидим в данной главе, основными конструктивными особенностями Spark сильно отличается от MapReduce. В отличие от Hadoop MapReduce, Spark не должен работать в тандеме с Apache Hadoop, хотя они часто и используются совместно. Spark унаследовал часть API, архитектуру и поддерживаемые форматы других существующих вычислительных платформ, в частности DryadLINQ[2]. Однако внутреннее устройство Spark, особенно обработка им ошибок, отличается от многих традиционных систем. Способность Spark реализовать отложенные вычисления в оперативной памяти уникальна. Создатели

---

[1] MapReduce — парадигма программирования, описывающая программы на языке процедур отображения (map), служащих для фильтрации и сортировки данных на узлах распределенной системы, и процедур свертки (reduce), агрегирующих данные на узлах отображения. Существуют реализации MapReduce на множестве языков, но это название обычно используют для популярной реализации Hadoop MapReduce (http://hadoop.apache.org/), идущей в одном пакете с распределенной файловой системой Apache Hadoop Distributed File System.

[2] DryadLINQ — исследовательский проект компании Microsoft, реализующий работу языка интегрированных запросов .NET (Language Integrated Query, LINQ) поверх механизма распределенного выполнения Dryad. Подобно Spark, в API DryadLINQ описывается объект, символизирующий распределенный набор данных, после чего предоставляются функции преобразования данных в виде определенных на объекте этого набора данных методов. DryadLINQ реализует отложенные вычисления, а его планировщик аналогичен планировщику Spark. Однако DryadLINQ не использует хранилище данных в оперативной памяти. Более подробную информацию вы можете найти в документации по DryadLINQ (https://www.microsoft.com/en-us/research/project/dryadlinq/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fprojects%2Fdryadlinq).

фреймворка считали его первым высокоуровневым языком программирования для быстрой распределенной обработки данных[1].

Чтобы извлечь максимум выгоды из Spark, важно понимать некоторые принципы его архитектуры и хотя бы на поверхностном уровне то, как выполняются программы фреймворка. В этой главе приводится широкий обзор модели распределенных вычислений Spark и подробное описание работы его планировщика и механизма выполнения. Мы будем ссылаться на изложенные здесь концепции во всей книге. Кроме того, надеемся, что эти пояснения обеспечат более четкое понимание каких-либо терминов, которые вы могли слышать от других пользователей Spark или встречать в документации по фреймворку.

Место Spark в экосистеме больших данных

Apache Spark — фреймворк с открытым исходным кодом, предоставляющий обобщаемые методы для параллельной обработки данных; одни и те же высокоуровневые функции Spark можно применять для выполнения несопоставимых по размеру задач обработки данных различного объема и структуры. Сам по себе фреймворк Spark не предназначен для хранения данных, проводимые им вычисления на JVM Spark (виртуальных машинах Java) сохраняются только на время жизни приложения Spark. Фреймворк может работать локально на отдельной машине с одной JVM (так называемый локальный режим). Но чаще он используется в сочетании с распределенной системой хранения данных (например, HDFS, Cassandra или S3) и диспетчером кластера для координации распределения приложений Spark по кластеру. В настоящий момент Spark поддерживает три вида диспетчеров кластера: автономный, Apache Mesos и Hadoop YARN (рис. 2.1). Автономный диспетчер кластера — часть фреймворка Spark, но его использование требует установки Spark во всех узлах кластера.

Компоненты Spark. Фреймворк предоставляет высокоуровневый язык запросов для обработки данных. У Spark Core, основного фреймворка обработки данных в экосистеме Spark, имеются API на Scala, Java, Python и R. Ядром Spark является абстракция данных под названием *Resilient Distributed Datasets* (RDD, отказоустойчивые распределенные наборы данных). Набор RDD — представление статически типизированных, распределенных коллекций с отложенным вычислением. У таких наборов имеется несколько предопределенных высокоуровневых преобразований (применяемых ко всему набору данных функций), например `map`, `join` и `reduce`, предназначенных для манипуляции распределенными наборами данных, а также функции ввода/вывода для чтения и записи данных между распределенной системой хранения и различными JVM Spark.

---

[1] См. первую статью о Spark (http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf) и другие посвященные ему материалы (http://spark.apache.org/research.html).

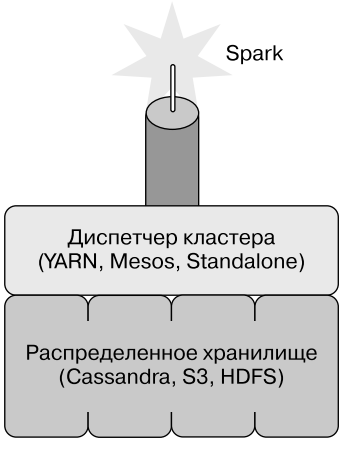

Рис. 2.1. Схема экосистемы обработки данных, включающей Spark

Хотя Spark и поддерживает язык R, в настоящее время RDD-интерфейс для этого языка отсутствует. Мы рассмотрим нюансы использования Java, Python, R и других языков подробнее в главе 7.

Помимо Spark Core экосистема Spark содержит несколько других компонентов от Apache, включая Spark SQL, Spark MLlib, Spark Streaming и GraphX[1], обеспечивающих более специализированные средства обработки данных. Некоторые из них объединяют со Spark Core одни и те же особенности в области производительности. Библиотека MLlib, к примеру, написана почти целиком на основе Spark API. Однако некоторые из компонентов выбиваются из этого ряда. У Spark SQL, например, не такой оптимизатор запросов, как у Spark Core.

Spark SQL — компонент, который можно использовать совместно со Spark Core, у него есть API для языков Scala, Java, Python и R и основных SQL-запросов. Spark SQL описывает интерфейс для частично структурированного типа данных `DataFrames` и, по состоянию на Spark 1.6, частично структурированную типизированную версию RDD — класс `Dataset`[2]. Spark SQL имеет значение для производительности Spark, и многое из выполняемого с помощью Spark Core можно сделать и благодаря Spark SQL. Мы рассмотрим последний подробнее в главе 3 и сравним производительность соединений в Spark SQL и Spark Core в главе 4.

[1] В настоящее время разработка компонента GraphX ведется не слишком активно, вероятно, он будет заменен GraphFrames или чем-то ему подобным.

[2] Типы данных Dataset и DataFrames были объединены в версии Spark 2.0. Datasets — это DataFrames объектов типа Row, к которым можно обращаться по номеру поля.

В фреймворке Spark есть два пакета машинного обучения: *ML* и *MLlib*. Второй — пакет алгоритмов для машинного обучения и статистики, написанный с помощью Spark. Spark ML все еще находится на ранней стадии разработки, существует начиная со Spark 1.2 и предоставляет более высокоуровневое API, чем MLlib, позволяющее пользователям упростить создание реальных конвейеров машинного обучения. Spark MLlib большей частью основывается на наборах RDD и задействует функции из Spark Core, в то время как в основе ML лежит класс `DataFrame`[1] Spark SQL. Сообщество разработчиков Spark планирует постепенно перейти к использованию ML и отказаться от MLlib. Соображения по производительности как Spark ML, так и MLlib отличаются от таковых для Spark Core и Spark SQL, мы обсудим их в главе 9.

Spark Streaming использует планировщик Spark Core для потоковой аналитики и мини-пакетов данных. Для Spark Streaming релевантны некоторые специфические соображения, например относительно размеров окон для пакетов. Мы приведем некоторые советы по использованию Spark Streaming в разделе «Организация потоковой обработки с помощью Spark» на с. 282.

GraphX — фреймворк для работы с графами, основанный на Spark и включающий API для вычислений на графах. GraphX — один из наименее «зрелых» компонентов Spark, так что мы не станем рассматривать его слишком детально. В будущих версиях Spark должны появиться механизмы для работы с типизированными графами поверх API Dataset. Мы вкратце познакомимся с GraphX в одноименном разделе главы 10.

В этой книге мы сосредоточимся на оптимизации программ, написанных с помощью Spark Core и Spark SQL. Однако, поскольку MLlib и другие фреймворки написаны с применением Spark API, мы рассмотрим инструменты, необходимые для их более эффективного использования. Возможно, к концу чтения книги вы уже будете готовы начать создавать собственные функции для MLlib и ML!

Помимо этих компонентов от Apache, сообществом разработчиков Spark было создано несколько библиотек, предоставляющих дополнительные возможности, например для тестирования или синтаксического разбора CSV-файлов, а также утилиты для подключения к различным источникам данных. На сайте http://spark-packages.org перечислено множество библиотек, которые можно динамически подключать во время выполнения с помощью команд `spark-submit` или `spark-shell` и добавлять в качестве зависимостей при компоновке к проектам `maven` или `sbt`. Сначала мы воспользуемся пакетами Spark для включения поддержки данных в формате CSV в пункте «Дополнительные форматы» на с. 80, а затем подробнее рассмотрим этот вопрос в разделе «Использование пакетов и библиотек, созданных сообществом разработчиков» на с. 297.

[1] См. документацию по пакету MLlib (http://spark.apache.org/docs/latest/mllib-guide.html).

Модель параллельных вычислений фреймворка Spark: наборы RDD

Spark позволяет пользователям писать программы для так называемого *драйвера* (ведущего узла) кластерной вычислительной системы, которая способна совершать операции над данными параллельно. Spark представляет большие наборы данных в виде наборов RDD — неизменяемых, распределенных коллекций объектов, — хранимых в *исполнителях* (ведомых узлах). Составляющие наборы объекты называются секциями и могут вычисляться (хотя и не обязательно) на различных узлах распределенной системы. Диспетчер кластера Spark запускает и распределяет исполнителей Spark по распределенной системе в соответствии с параметрами конфигурации, задаваемыми приложением Spark. Механизм выполнения Spark сам распределяет данные по исполнителям для совершения вычислений (см. рис. 2.4 ниже).

Вместо вычисления каждого преобразования сразу же после того, как его задаст драйверная программа, Spark вычисляет RDD отложенным образом, рассчитывая результат преобразований набора только в момент, когда становятся необходимы итоговые данные RDD (зачастую для записи в хранилище или отправки агрегированных данных драйверу). Spark умеет хранить RDD в оперативной памяти узлов-исполнителей во время всего времени жизни приложения Spark ради ускорения доступа при повторяемых вычислениях. Наборы RDD реализованы в Spark как неизменяемые, так что преобразование объекта RDD возвращает новый объект, а не уже существующий. Как мы увидим в этой главе, подобная парадигма отложенного вычисления, хранения данных в оперативной памяти, а также неизменяемости делает Spark удобным в использовании, отказоустойчивым, масштабируемым и эффективным.

Отложенные вычисления

В основе множества других систем, хранящих данные в оперативной памяти, лежат «мелкозернистые» обновления изменяемых объектов, то есть обращения к конкретной ячейке таблицы для сохранения промежуточных результатов. Напротив, набор RDD вычисляется полностью отложенным образом. Spark приступает к вычислению секций лишь при вызове действия.

Действие — операция Spark, возвращающая что-либо внешней по отношению к Spark (к исполнителям Spark) системе. Примером может послужить возврат данных драйверу (такими операциями, как `count` или `collect`) или запись данных во внешнюю систему хранения (например, операцией `copyToHadoop`). Действия вызывают срабатывание планировщика, который строит *ориентированный ациклический граф* (directed acyclic graph, DAG) на основе зависимостей между преобразованиями

наборов RDD. Другими словами, Spark вычисляет действие в обратном порядке путем описания последовательности шагов, необходимых для формирования каждого из объектов итогового распределенного набора данных (каждой секции). А затем согласно этой последовательности, называемой планом выполнения, планировщик находит недостающие секции для каждого этапа до тех пор, пока не вычислит весь результат.

Далеко не все преобразования являются на 100 % отложенными. Преобразованию sortByKey требуется вычислить RDD, чтобы определить диапазон данных, поэтому оно включает в себя и преобразование, и действие.

Преимущества отложенного вычисления с точки зрения производительности и удобства использования

Отложенное вычисление позволяет Spark объединять операции, не требующие взаимодействия с драйвером (их называют преобразованиями с взаимно однозначными зависимостями), чтобы избежать многократных проходов по данным. Например, пусть Spark-программа вызывает для одного и того же набора RDD функции `map` и `filter`. Spark может отправить каждому из исполнителей инструкции по выполнению как `map`, так и `filter`. Далее фреймворк может выполнить обе эти функции для каждой из секций, что потребует лишь однократного обращения к данным вместо отправки двойного набора инструкций и обращения к каждой секции два раза. Теоретически это должно вдвое снизить вычислительную сложность.

Парадигма отложенных вычислений не только более эффективна, но и позволяет легче реализовать в Spark (по сравнению с другими фреймворками, например MapReduce) логику, требующую от разработчика объединения операций отображения. Разумная стратегия отложенных вычислений фреймворка Spark обеспечивает возможность выражения той же логики намного меньшим количеством строк кода. Для этого достаточно выстроить в цепочки операции с узкими зависимостями, после чего механизм вычисления Spark выполнит всю работу по их объединению.

Рассмотрим классический пример подсчета количества вхождений слов в заданный набор документов: сначала производится разбор текста на слова, после чего вычисляется количество вхождений для каждого слова. В документации Apache приведен образец подсчета слов, который даже в простейшей форме занимает около 50 строк кода (https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html#Example%3A+WordCount+v1.0) (включая операторы импорта) на языке программирования Java. Сравнимая реализация с помощью Spark, доступная на сайте Apache (http://spark.apache.org/examples.html), занимает приблизительно 15 строк кода на Java и пять — на Scala. Приведенный пример включает этапы чтения данных, создание карты слов по документам и подсчет количества слов. Мы воспроизвели его в примере 2.1.

Пример 2.1. Простейший пример подсчета количества слов

```
def simpleWordCount(rdd: RDD[String]): RDD[(String, Int)] = {
  val words = rdd.flatMap(_.split(" "))
  val wordPairs = words.map((_, 1))
  val wordCounts = wordPairs.reduceByKey(_ + _)
  wordCounts
}
```

Дальнейшие преимущества реализации подсчета количества слов на Spark заключаются в упрощении ее модификации и усовершенствовании. Допустим, что мы хотели бы модифицировать данную функцию и отфильтровать определенные стоп-слова и знаки препинания из всех документов, прежде чем подсчитывать количество вхождений слов. В MapReduce для этого нужно было бы добавить в подпрограмму отображения логику фильтрации, чтобы избежать второго прохода по данным. Реализацию этой программы на MapReduce можно найти вот здесь: https://github.com/kite-sdk/kite/wiki/WordCount-Version-Three. Для соответствующей модификации программы на Spark, напротив, достаточно добавить шаг `filter` перед создающим пары «ключ — значение» шагом `map`. Пример 2.2 демонстрирует объединение шагов `map` и `filter` с помощью методики отложенного вычисления Spark.

Пример 2.2. Подсчет количества слов с фильтрацией стоп-слов

```
def withStopWordsFiltered(rdd : RDD[String], illegalTokens : Array[Char],
  stopWords : Set[String]): RDD[(String, Int)] = {
  val separators = illegalTokens ++ Array[Char](' ')
  val tokens: RDD[String] = rdd.flatMap(_.split(separators).map(_.trim.toLowerCase))
  val words = tokens.filter(token =>
    !stopWords.contains(token) && (token.length > 0) )
  val wordPairs = words.map((_, 1))
  val wordCounts = wordPairs.reduceByKey(_ + _)
  wordCounts
}
```

Отложенное вычисление и отказоустойчивость

Spark отличается отказоустойчивостью. Это значит, что он не подведет, не потеряет данные и не вернет неправильные результаты даже в случае отказа машины, на которой он работает, или сбоя сети. Уникальность обеспечения отказоустойчивости фреймворком Spark состоит в следующем: каждая секция данных содержит информацию о зависимостях, достаточную для повторного ее вычисления. Большинство парадигм распределенных вычислений, при которых пользователи могут работать с изменяемыми объектами, обеспечивают отказоустойчивость за счет журналирования обновлений или дублирования данных на других машинах.

В отличие от них Spark не требует поддержания журнала обновлений каждого объекта RDD или журнала промежуточных шагов, поскольку сам объект содержит всю информацию о зависимостях, необходимую для репликации всех его секций. Следовательно, при потере секции в RDD есть достаточно информации относительно ее происхождения, чтобы вычислить ее заново, причем этот расчет можно распараллелить для ускорения восстановления.

Отложенное вычисление и отладка

Последствия отложенного вычисления для отладки очень существенны, ведь оно означает возможность сбоя Spark-программы только в момент выполнения действия. Допустим, вы работали с примером подсчета количества слов, после чего аккумулировали результаты на драйвере. Если переданное значение для стоп-слов было неопределенным (например, из-за того, что оно представляло собой результат выполнения программы на языке Java), конечно, произойдет сбой обработки кода с исключением «указатель, содержащий неопределенное значение» в функции проверки `contains`. Однако подобный сбой проявится, только когда программа начнет выполнять этап сбора результатов. Даже отслеживание стека будет показывать: сбой произошел на этапе сбора, наводя на мысль о том, что источник его — в операторе сбора. Поэтому лучше всего вести разработку в среде, предоставляющей доступ к полной отладочной информации.

Из-за отложенного вычисления отслеживание вызовов сбойных заданий Spark в стеке (особенно при внедрении в бóльшие системы) будет зачастую неизменно указывать на сбой в момент действия, даже если проблема в логике возникла в преобразовании, происходящем намного раньше по ходу программы.

Хранение данных в памяти и управление памятью

Превосходство Spark над MapReduce в производительности особенно ощутимо в сценариях использования, включающих повторяемые вычисления. В значительной степени фреймворк обязан этим повышением производительности хранению данных в оперативной памяти. Вместо записи данных на диск в промежутке между проходами Spark имеет возможность хранить данные в исполнителях, загруженных в оперативную память. При этом данные каждой секции доступны в оперативной памяти всякий раз, когда они нужны.

Spark предлагает три варианта управления памятью: в оперативной памяти в виде сериализованных данных, в оперативной памяти в виде десериализованных данных и на жестком диске. У каждого из них есть свои преимущества относительно времени выполнения и занимаемого пространства.

- *В оперативной памяти в виде десериализованных Java-объектов.* Наиболее очевидный способ хранения объектов в наборе RDD — в виде исходных десериализованных Java-объектов, определяемых драйверной программой. Это самая быстродействующая форма хранения данных в оперативной памяти, поскольку экономит требуемое на сериализацию время. Однако она, возможно, не самая выгодная относительно используемой памяти, так как данные приходится хранить в виде объектов.
- *В виде сериализованных данных.* При перемещении по сети объекты Spark преобразуются в потоки байтов с помощью стандартной библиотеки сериализации

языка Java. Это, вероятно, более медленно работающий подход, поскольку чтение сериализованных данных требует от CPU большего объема действий, чем чтение десериализованных. Однако это зачастую выгоднее в смысле используемой памяти, поскольку пользователь получает возможность выбирать наиболее эффективное представление. В то время как Java-сериализация более эффективна, чем использование целых объектов, Kryo-сериализация (которую мы обсудим в подразделе «Kryo» на с. 317), возможно, еще эффективнее в смысле расхода памяти.

- *На диске*. Наборы RDD, секции которых слишком велики для хранения в оперативной памяти каждого из исполнителей, можно записать на диск. Это явно более медленный метод в случае повторяемых вычислений, но, вероятно, и более отказоустойчивый при длинных последовательностях преобразований и, наверное, единственный разумный вариант при очень больших объемах вычислений.

Функция `persist()` классов RDD позволяет пользователю контролировать метод хранения набора RDD. По умолчанию `persist()` сохраняет набор RDD в виде десериализованных объектов в оперативной памяти, но пользователь может управлять способом хранения RDD путем передачи в `persist()` в виде параметра одной из множества опций хранения. Мы рассмотрим различные варианты повторного использования набора RDD в подразделе «Виды повторного использования: кэш, сохранение, контрольная точка, перетасовочные файлы» на с. 140. При сохранении наборов RDD их реализация по умолчанию вытесняет наиболее давно применяемую секцию (так называемое LRU-кэширование), если место, которое она занимает, необходимо для вычисления или кэширования новой секции. Однако это поведение можно изменить, управляя назначением приоритетов с помощью функции `persistencePriority()` классов RDD (см. подраздел «LRU-кэширование» на с. 145).

Неизменяемость и интерфейс RDD

Фреймворк Spark определяет интерфейс RDD с помощью свойств, которые должен реализовывать любой тип RDD. Эти свойства включают зависимости RDD и информацию о локальности данных, необходимые механизму выполнения для вычисления набора RDD. В силу неизменяемости и статической типизации RDD вызов преобразования для RDD не меняет исходный объект RDD, а приводит к возврату нового объекта с новым описанием свойств RDD.

Создавать наборы RDD можно тремя способами:

- путем преобразования существующего RDD;
- из объекта `SparkContext` — API-шлюза вашего приложения к Spark;
- путем преобразования типа объектов `DataFrame` или `Dataset` (созданного из `SparkSession`[1]).

[1] В версиях Spark, более ранних, чем 2.0, класс SparkSession носил название SQLContext.

Объект `SparkContext` олицетворяет соединение между кластером Spark и одним из запущенных приложений Spark. Можно использовать его для создания RDD из локального объекта Scala (с применением одного из методов `makeRDD` или `parallelize`) или с помощью чтения данных из устойчивого хранилища (текстовые файлы, двоичные файлы, Hadoop Context или файл Hadoop). Объекты `DataFrame` и `Dataset` можно читать, задействуя `SparkSession` — объект Spark SQL, эквивалентный `SparkContext`.

Для внутренних целей Spark использует для представления RDD пять основных свойств. Три обязательных свойства: список объектов секций, составляющих набор RDD, функция вычисления итератора секции и список зависимостей от других наборов. Кроме того, объекты RDD могут также включать объект `partitioner` (для RDD, состоящих из строк пар «ключ — значение», представленных в виде кортежей Scala) и список предпочтительных местоположений (для HDFS-файлов). Вам, как конечному пользователю, редко будут нужны все пять свойств, скорее вы станете задействовать предопределенные преобразования RDD. Однако полезно (для отладки и общего представления) знать эти свойства и уметь к ним обращаться. Пять свойств соответствуют следующим доступным конечному пользователю (то есть вам) пяти методам.

- `partitions()` — возвращает массив объектов секций, составляющих части распределенного набора данных. Если речь идет об RDD с объектом `partitioner`, то значение индекса каждой из секций будет соответствовать значению, возвращаемому функцией `getPartition` для каждого ключа данных, относящегося к этой секции.
- `iterator(p, parentIters)` — вычисляет элементы секции `p` с помощью заданных итераторов для каждой из ее родительских секций. Эта функция вызывается для вычисления каждой из секций данного RDD. Она не предназначена для непосредственного вызова пользователем, а применяется фреймворком Spark при вычислении действий. Тем не менее может быть полезно заглянуть в реализацию этой функции, чтобы понять, как вычисляется каждая из секций преобразования RDD.
- `dependencies()` — возвращает последовательность объектов зависимостей. Благодаря зависимостям планировщик знает, как данный RDD зависит от других. Существует два вида зависимостей: *узкие* (объекты `NarrowDependency`), соответствующие секциям, зависящим от одной или небольшого количества родительских секций, и *широкие* (объекты `ShuffleDependency`), используемые в случаях, когда для вычисления секции необходима перегруппировка всех данных из родительских секций. Мы обсудим эти типы зависимостей в подразделе «Широкие и узкие зависимости» далее в текущем разделе.
- `partitioner()` — возвращает вариантный тип Scala для объекта `partitioner`, если между `element` и `partition` объекта RDD есть связанная с ним функция, например `hashPartitioner`. Эта функция возвращает `None` для всех наборов RDD, чей тип не кортеж (которые не представляют данные типа «ключ — значение»). У представляющего HDFS-файл набора (реализован в *NewHadoopRDD.scala*) каждому блоку файла соответствует секция. Мы обсудим секционирование подробнее в подразделе «Использование объекта Partitioner фреймворка Spark» на с. 167.

- `preferredLocations(p)` — возвращает информацию о локальности данных секции `p`. Точнее говоря, эта функция возвращает последовательность строк с информацией о каждом из узлов, на которых хранятся фрагменты `p`. В случае представляющего HDFS-файл набора RDD каждая строка возвращаемого функцией `preferredLocations` результата представляет собой Hadoop-имя узла, в котором хранится данная секция.

Типы наборов RDD

Реализация API Spark Scala содержит абстрактный класс `RDD`, который заключает в себе не только пять базовых функций RDD, но и доступные для всех RDD преобразования и действия, например `map` и `collect`. Функции, определенные только для RDD конкретного типа, описываются в нескольких классах функций RDD, в том числе `PairRDDFunctions`, `OrderedRDDFunctions` и `GroupedRDDFunctions`. Видимость дополнительных функций из этих классов обеспечивается путем неявного преобразования типов из абстрактного класса `RDD` на основе информации о типе или при применении к RDD преобразования.

API Spark также содержит реализации класса `RDD`, в которых путем переопределения базовых свойств RDD задаются и более конкретные варианты поведения. Среди них уже обсуждавшийся ранее класс `NewHadoopRDD`, представляющий набор RDD, созданный из HDFS-файловой системы, и класс `ShuffledRDD`, представляющий уже секционированный набор. В каждой из этих реализаций RDD есть свойства, относящиеся именно к RDD данного типа. Создание объекта RDD с помощью преобразования или `SparkContext` приведет к возврату экземпляра одной из этих реализаций класса RDD. Сигнатура некоторых операций RDD в языке Java отличается от их сигнатуры в языке Scala. Они описаны в классе `JavaRDD.java`.

Узнать тип RDD можно с помощью функции toDebugString, определенной во всех классах RDD. Она возвращает информацию о типе RDD и предоставляет список всех его родительских RDD-классов.

Мы обсудим подробнее различные типы RDD и преобразования RDD в главах 5 и 6.

Функции, применяемые к наборам RDD: преобразования и действия

Существует два типа определяемых для наборов RDD функций: *действия* (actions) и *преобразования* (transformations). Действия — это функции, возвращающие нечто отличное от набора RDD, включая побочный эффект, а преобразования — функции, возвращающие другой набор.

В каждой программе Spark должно содержаться действие, поскольку действия или передают информацию обратно драйверу, или записывают данные в устойчивое хранилище. Именно действия инициируют вычисление для программы Spark. Вызовы `persist` тоже инициируют вычисление, но обычно не отмечают завершение задания Spark. Действия, передающие информацию обратно драйверу: `collect`, `count`, `collectAsMap`, `sample`, `reduce` и `take`.

Некоторые из этих действий плохо масштабируются, так как могут приводить к ошибкам памяти в драйвере. В целом лучше использовать действия take, count и reduce, передающие обратно драйверу фиксированный объем данных, а не collect или sample.

Среди действий, выполняющих запись в хранилище, — `saveAsTextFile`, `saveAsSequenceFile` и `saveAsObjectFile`. Большинство действий, сохраняющих данные в Hadoop, доступны только для RDD-наборов пар «ключ — значение»; они определены как в классе `PairRDDFunctions` (предоставляющем путем неявного преобразования типов методы для RDD-кортежей), так и в классе `NewHadoopRDD` — реализации для RDD, созданных с помощью чтения данных из Hadoop. Отдельные функции сохранения, например `saveAsTextFile` и `saveAsObjectFile`, доступны для всех RDD. Они работают путем добавления к каждой из записей неявного пустого ключа (который затем игнорируется при сохранении). Функции, ничего не возвращающие (`void` в языке Java или `Unit` в языке Scala), такие как `foreach`, тоже являются действиями: они инициируют выполнение задания Spark. Функцию `foreach` можно использовать для запуска вычисления RDD, но ее часто применяют и для записи данных в неподдерживаемые форматы (например, в конечные веб-точки).

Большинство возможностей API Spark связано с преобразованиями. Spark-преобразования — это высокоуровневые преобразования, используемые для сортировки, свертки, группировки, выборки, фильтрации и отображения распределенных данных. Мы обсудим преобразования подробнее в главе 6, посвященной исключительно преобразованиям RDD пар «ключ — значение», а также в главе 5.

Широкие и узкие зависимости

Самая важная информация, которую нужно знать о преобразованиях для понимания механизма вычисления наборов RDD, — то, что они делятся на две категории: преобразования с *узкими зависимостями* (narrow dependencies) и преобразования с *широкими зависимостями* (wide dependencies). Различие между узкими и широкими преобразованиями влечет достаточно важные последствия для способа вычисления преобразования и, следовательно, производительности. Мы дадим формальное определение узких и широких преобразований, чтобы лучше понять парадигму выполнения Spark, в разделе «Планирование заданий Spark» далее в этой главе, но более подробное пояснение связанных с ними соображений производительности отложим до главы 5.

По существу, узкие преобразования — это те, в которых у всех секций дочернего RDD имеются простые, конечные зависимости от секций родительского. Зависимости являются узкими только в том случае, если их можно определить во время проектирования, без привязки к значениям записей в родительских секциях, и если у каждой родительской секции есть хотя бы одна дочерняя. Точнее говоря, секции при узких преобразованиях могут зависеть или от одного родителя (например, как в операторе `map`), или от уникального подмножества родительских секций, определенного уже во время проектирования (`coalesce`). Следовательно, узкие преобразования можно выполнять на произвольном подмножестве данных, не имея никакой информации о других секциях. Напротив, преобразования с широкими зависимостями нельзя выполнять на произвольных строках, они требуют секционирования данных определенным образом, допустим, в соответствии со значениями их ключей. В преобразовании `sort`, например, записи должны быть секционированы так, чтобы ключи из одного диапазона находились в одной секции. Преобразования с широкими зависимостями включают `sort`, `reduceByKey`, `groupByKey`, `join`, а также все преобразования, вызывающие функцию `rePartition`.

В некоторых случаях, например, когда Spark заранее знает, что данные секционированы определенным образом, операции с широкими зависимостями не приводят к перетасовке. Если операция требует выполнения перетасовки, фреймворк добавляет в список зависимостей, относящихся к данному набору RDD, объект `ShuffledDependency`. В целом перетасовки довольно затратны. Расходы на них повышаются при увеличении объемов данных и в случаях необходимости перемещения, во время перетасовки, *б*ольшей доли этих данных в другую секцию. Как мы увидим в главе 6, можно добиться значительного роста производительности программ Spark, если делать меньше перетасовок, причем менее затратных.

Следующие две схемы иллюстрируют различие в графах зависимостей для преобразований с узкими и широкими зависимостями. На рис. 2.2 показаны узкие зависимости, в которых каждая дочерняя секция (каждый из квадратов в нижней строке) зависит от известного подмножества родительских секций. Узкие зависимости изображены стрелками. Слева показан граф зависимостей для узких преобразований (например, `map`, `filter`, `mapPartitions` или `flatMap`). Справа вверху представлены зависимости между секциями для операции `coalesce`. В этом случае мы пытаемся проиллюстрировать то, что преобразование может по-прежнему считаться узким, даже если дочерние секции зависят от нескольких родительских, при наличии возможности установить множество родительских секций независимо от содержащихся в секциях значений данных.

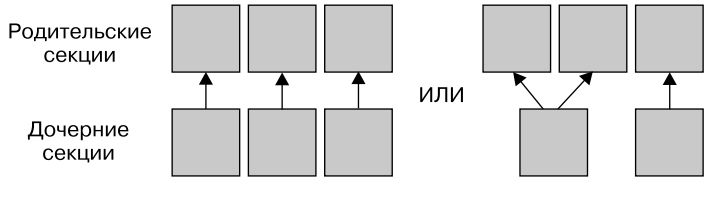

Рис. 2.2. Простая схема зависимостей между секциями в случае узких преобразований

Рисунок 2.3 демонстрирует широкие зависимости между секциями. В этом случае дочерние секции (показанные внизу) зависят от произвольного множества родительских секций. Широкие зависимости (показанные стрелками) неизвестны полностью до вычисления данных. В отличие от операции `coalesce` данные секционированы в соответствии с их значениями. Указанному паттерну следуют графы зависимостей любых требующих перетасовок операций (например, `groupByKey`, `reduceByKey`, `sort` и `sortByKey`).

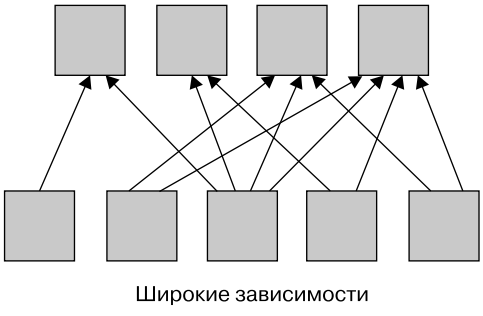

Рис. 2.3. Простая схема зависимостей между секциями в случае широких преобразований

Функции соединений несколько более сложны в этом случае, поскольку у них могут быть как узкие, так и широкие зависимости, что определяется способом секционирования двух родительских наборов RDD. Мы проиллюстрируем зависимости в различных сценариях операций соединения в разделе «Соединения Spark Core» главы 4.

Планирование заданий Spark

Приложение Spark состоит из ведущего процесса, содержащего высокоуровневую логику Spark, и набора процессов-исполнителей, которые могут быть разбросаны по узлам кластера. Сама программа Spark работает на узле драйвера и отправляет команды исполнителям. На одном кластере фреймворка можно запустить в конкурентном режиме несколько приложений Spark. Диспетчер кластера планирует выполнение приложений, каждое из которых соответствует одному `SparkContext`. Приложения Spark, в свою очередь, запускают множество конкурентных заданий. Эти задания соответствуют действиям над набором RDD в конкретном приложении. В данном разделе мы рассмотрим приложение Spark и запуск им заданий Spark: процессов вычисления преобразований RDD.

Выделение ресурсов приложениям

Spark предоставляет два способа выделения ресурсов приложениям: *статическое* (static allocation) и *динамическое* (dynamic allocation). При статическом каждому приложению выделяется конечный объем ресурсов кластера, который сохраняется

за ним на протяжении всей жизни приложения (на время работы SparkContext). В зависимости от кластера к категории статического выделения относится множество различных видов выделения ресурсов. Более подробную информацию можно найти в документации Spark по планированию заданий (http://spark.apache.org/docs/latest/job-scheduling.html).

Начиная с версии 1.2, фреймворк Spark позволяет динамически выделять ресурсы, расширяя возможности статического выделения. При динамическом выделении исполнители добавляются в приложение Spark и удаляются из него по мере необходимости, на основе набора эвристических правил для ожидаемой потребности в ресурсах. Мы обсудим выделение ресурсов подробнее в подразделе «Выделение ресурсов кластера и динамическое выделение» на с. 308.

Приложение Spark

Приложение Spark соответствует набору заданий Spark, определяемому одним объектом SparkContext в драйверной программе. Приложение Spark запускается в момент начала выполнения SparkContext. При выполнении SparkContext в рабочих узлах кластера запускается драйвер и набор исполнителей. Каждый исполнитель соответствует виртуальной машине Java (JVM), так что не способен охватывать несколько узлов, хотя в одном узле может быть несколько исполнителей.

Объект SparkContext определяет, сколько ресурсов выделяется каждому исполнителю. При запуске задания Spark каждый исполнитель получает слоты для запуска задач, необходимых для вычисления RDD. Таким образом, можно рассматривать SparkContext как набор параметров конфигурации для выполнения заданий Spark. Эти параметры доступны через объект SparkConf, применяемый для создания объекта SparkContext. Мы обсудим, как задействовать эти параметры, в приложении, приведенном в конце книги. Одно приложение часто, хотя и не всегда, соответствует одному пользователю. То есть каждая работающая в вашем кластере программа Spark, вероятно, применяет один SparkContext.

RDD не могут использоваться приложениями совместно. Следовательно, у преобразований, например join, задействующих более одного RDD, должен быть один и тот же SparkContext.

Рисунок 2.4 иллюстрирует, что происходит при запуске SparkContext. Во-первых, драйверная программа производит тестовый опрос диспетчера кластера. Диспетчер кластера запускает несколько исполнителей Spark (JVM, показанных на схеме прямоугольниками) в рабочих узлах кластера (показаны в виде окружностей). В одном узле может быть несколько исполнителей Spark, но у исполнителя не получится охватывать несколько узлов. Набор RDD будет вычисляться в секциях (показанных серыми овалами) по исполнителям. Каждый исполнитель может иметь несколько секций, но секция не может распространяться на несколько исполнителей.

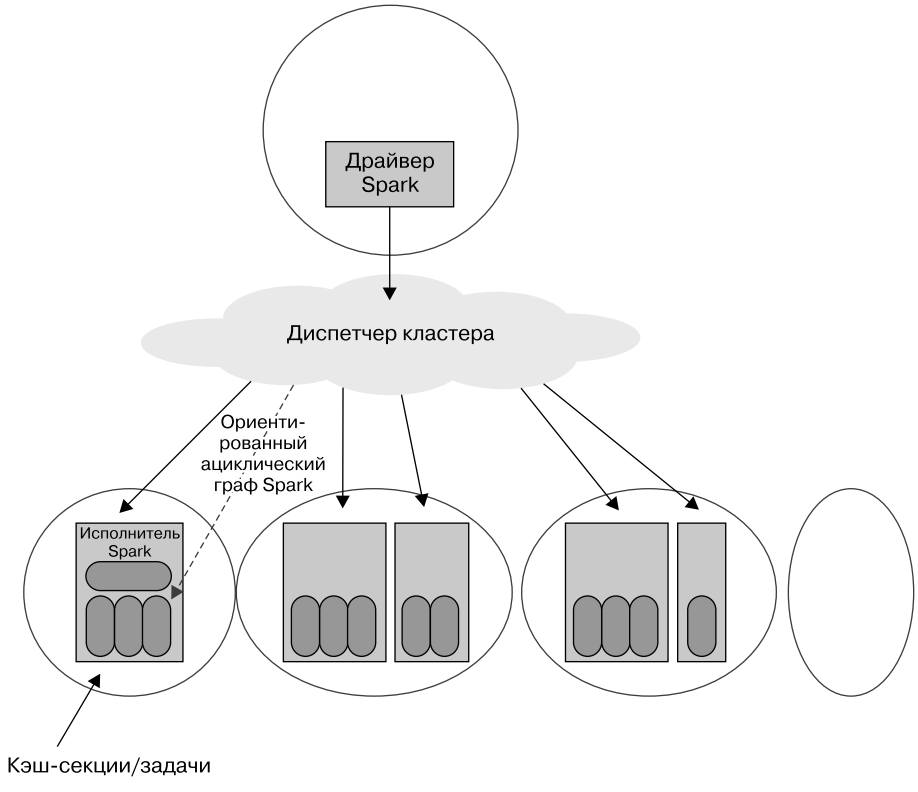

Рис. 2.4. Запуск приложения Spark в распределенной системе

Планировщик Spark по умолчанию. По умолчанию фреймворк планирует задания на основе принципа «первым вошел, первым вышел». Однако Spark предоставляет недискриминационный планировщик, который назначает задачи конкурентным заданиям циклическим образом, то есть выделяя по нескольку задач каждому заданию вплоть до завершения всех заданий. Данный планировщик гарантирует получение заданиями справедливой доли ресурсов кластера. После этого приложение Spark запускает задания в порядке, соответствующем вызовам действий в `SparkContext`.

Структура задания Spark

В соответствии с парадигмой отложенного вычисления фреймворка приложение Spark «ничего не делает» до тех пор, пока драйверная программа не вызовет действие. При каждом вызове действия планировщик Spark строит граф выполнения и запускает *задание Spark* (Spark job). Оно состоит из *этапов*, представляющих собой шаги преобразования данных, необходимые для формирования итогового

набора RDD. Этап состоит из набора *задач* (tasks), каждая из которых означает параллельное вычисление, выполняемое на исполнителе.

На рис. 2.5 показано дерево различных компонентов приложения Spark и их соответствие вызовам API. Работа приложения соответствует запуску `SparkContext/SparkSession`. *Приложение* (application) может содержать много заданий, каждое из которых соответствует одному действию RDD. *Задание* (job) может состоять из нескольких этапов, каждый из которых соответствует широкому преобразованию. *Этап* (stage) состоит из одной или нескольких задач, каждая из которых соответствует параллелизуемой единице вычислений, выполняемой на данном этапе. Секции в итоговом RDD данного этапа соответствует одна *задача* (task).

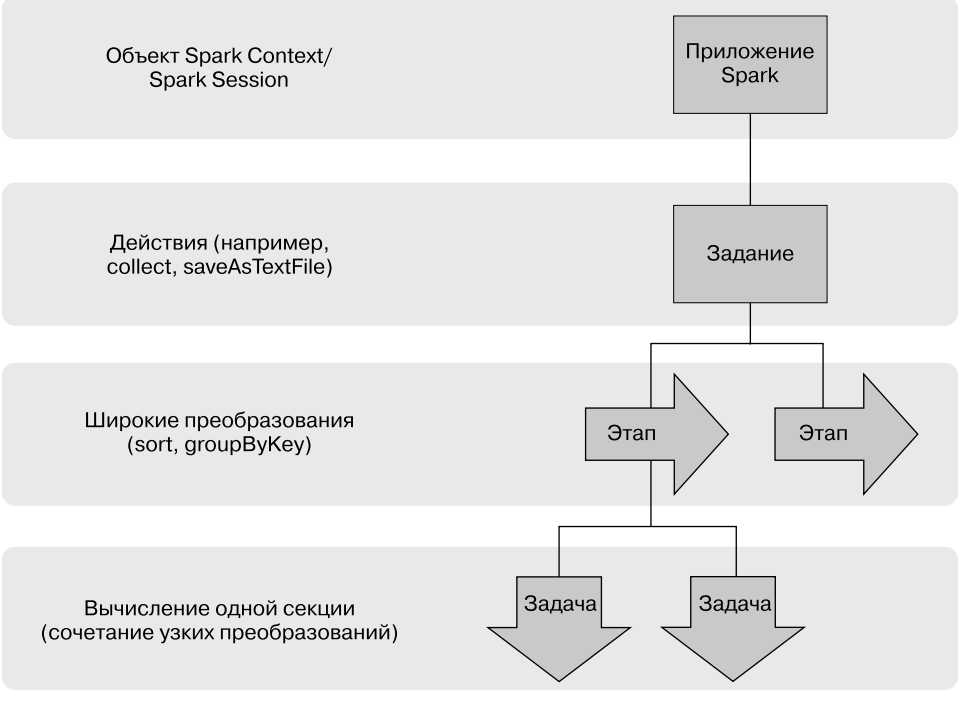

Рис. 2.5. Дерево приложения Spark

DAG

Высокоуровневый слой планирования Spark использует зависимости RDD для построения *ориентированного ациклического графа* (directed acyclic graph, DAG) этапов для каждого задания Spark. В API Spark это носит название планировщика DAG (DAG Scheduler). Как вы, вероятно, заметили, ошибки, связанные с подключением к кластеру, параметрами конфигурации и запуском заданий Spark

отображаются в виде ошибок планировщика DAG. Так происходит потому, что выполнением заданий Spark занимается DAG. Он строит граф этапов для каждого задания, определяет, где должна выполняться каждая из задач, и передает данную информацию `TaskScheduler`, отвечающему за выполнение задач в кластере. `TaskScheduler` создает граф зависимостей между секциями[1].

Задания

Задание Spark — высший элемент иерархии выполнения фреймворка. Каждое задание соответствует одному действию, а действия вызываются драйверной программой приложения Spark. Как мы уже обсуждали в подразделе «Функции над наборами RDD: преобразования и действия» на с. 35, действие, в частности, можно представить как нечто перемещающее данные из «RDD-мира Spark» в некую другую систему хранения (обычно передавая данные драйверу или записывая в другую устойчивую систему хранения).

Ребра графа выполнения Spark формируются на основе зависимостей между секциями в преобразованиях наборов RDD (как показано на рис. 2.2 и 2.3). Следовательно, у операции, возвращающей нечто отличное от набора, не может быть дочерних элементов. Язык теории графов позволяет сказать, что это действие формирует «лист» в DAG. Следовательно, с одним графом выполнения можно связать произвольно большой набор преобразований. Однако сразу же после вызова действия Spark теряет возможность добавлять элементы в данный граф. Приложение запускает задание, включающее преобразования, необходимые для вычисления вызвавшего действие итогового набора RDD.

Этапы

Напомним, что фреймворк Spark осуществляет отложенное вычисление преобразований; последние не вычисляются до тех пор, пока не будет вызвано действие. Как уже упоминалось, задание определяется путем вызова действия. Оно может включать одно или несколько преобразований, причем широкие преобразования определяют разбиение заданий на *этапы* (stages).

Каждому этапу соответствует создаваемая широким преобразованием в программе Spark перетасовочная зависимость. На высоком уровне этап можно рассматривать как набор вычислений (задач), каждое из которых выполняется одним исполнителем без взаимодействия с другими исполнителями или драйвером. Иными словами, граница нового этапа определяется необходимостью сетевого взаимодействия между рабочими узлами, например при перетасовке.

---

[1] См. более подробное описание TaskScheduler на https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-TaskScheduler.html.

Такие зависимости, формирующие границы этапов, называются перетасовочными (`ShuffleDependencies`). Как мы уже упоминали в подразделе «Широкие и узкие зависимости» на с. 36, к перетасовкам приводят те из широких преобразований, которые требуют перераспределения данных по секциям, например как `sort` и `groupByKey`. Несколько преобразований с узкими зависимостями можно сгруппировать в один этап.

Как мы видели в образце подсчета количества вхождений слов, в котором фильтровали стоп-слова (см. пример 2.2), фреймворк Spark умеет объединять шаги `flatMap`, `map` и `filter` в один этап, поскольку ни одно из указанных преобразований не требует перетасовки. Следовательно, все исполнители могут применять шаги `flatMap`, `map` и `filter` последовательно, за один проход данных.

Spark отслеживает секционирование наборов RDD, так что не требует секционирования одного RDD на основе одного объекта Partitioner более одного раза. Это влечет интересные последствия для DAG: одни и те же операции над набором RDD с известными способами секционирования и RDD без таковых могут приводить к разным границам этапов, поскольку RDD с имеющейся информацией о секционировании нужды перетасовывать нет (а значит, последующие преобразования относятся к тому же этапу). Последствия наличия информации о секционировании для вычислительного процесса мы обсудим в главе 6.

Поскольку границы этапов требуют взаимодействия с драйвером, то относящиеся к одному заданию этапы обычно приходится выполнять последовательно, а не параллельно. Последнее возможно, если этапы используются для вычисления различных наборов RDD, объединяемых в обратное преобразование, например `join`. Однако необходимое для вычисления одного набора RDD широкое преобразование нужно вычислять последовательно. Исходя из вышесказанного, обычно желательно проектировать программы так, чтобы требовалось меньшее количество перетасовок.

Задачи

Этапы состоят из задач. *Задача* (task) — самый маленький блок иерархии выполнения, каждая задача соответствует одному локальному расчету. Все задачи одного этапа выполняет один и тот же код, но для различных элементов данных. Одна задача не может выполняться более чем в одном исполнителе. Однако у каждого исполнителя есть динамически выделяемое количество слотов для выполнения задач, и он способен выполнять несколько задач параллельно в течение своего жизненного цикла. Количество задач этапа соответствует количеству секций выходного набора RDD данного этапа.

Рисунок 2.6 демонстрирует вычисление задачи Spark, которая является результатом драйверной программы, вызывающей простую программу Spark, показанную в примере 2.3.

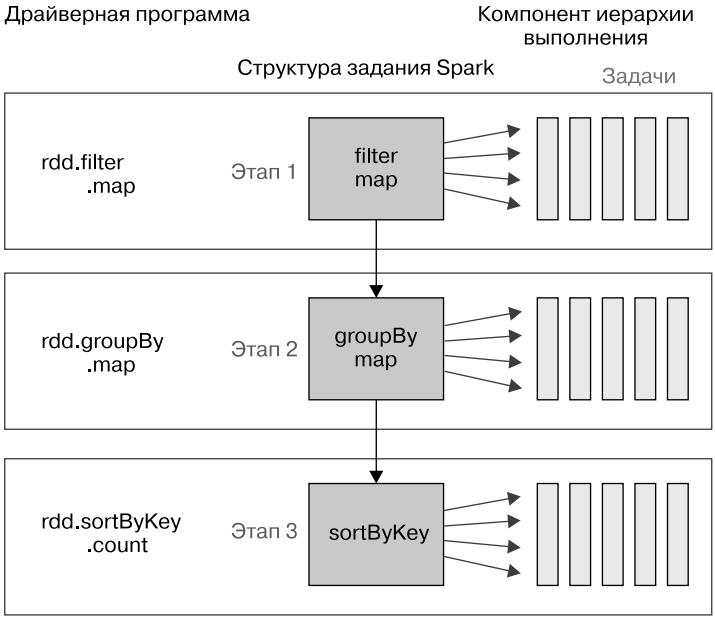

Рис. 2.6. Схема этапов для простой программы Spark, показанной в примере 2.3

Пример 2.3. Демонстрация границ этапов для различных типов преобразований

```
def simpleSparkProgram(rdd : RDD[Double]): Long ={
// Этап 1
  rdd.filter(_< 1000.0)
     .map(x => (x, x) )
// Этап 2
     .groupByKey()
     .map{ case(value, groups) => (groups.sum, value)}
// Этап 3
     .sortByKey()
     .count()
}
```

Этапы (темные прямоугольники) ограничиваются операциями перетасовки `groupByKey` и `sortByKey`. Каждый этап состоит из нескольких задач, выполняемых параллельно: по одной для каждой секции в получаемом результате преобразований RDD (показаны в виде светлых прямоугольников).

Кластер не обязательно выполняет все задачи для каждого этапа параллельно. У каждого исполнителя есть несколько ядер. Количество ядер в расчете на исполнителя настраивается на уровне приложения, но обычно соответствует количеству физических ядер кластера[1]. Количество одновременно выполняемых

[1] См. информацию о настройке количества ядер и соотношениях между количеством ядер Spark и CPU в разделе «Основные настройки Spark Core: сколько ресурсов нужно выделять приложению Spark» приложения.

фреймворком задач не может превышать общее количество ядер исполнителей, выделенных приложению. Максимальное количество заданий можно рассчитать на основе параметров из Spark Conf по следующей формуле: общее количество ядер исполнителей = количество ядер в расчете на исполнителя × количество исполнителей. Если количество секций (а значит, и количество задач) превышает количество слотов для выполнения задач, то лишние задачи будут назначены исполнителям после завершения первого цикла выполнения задач и освобождения ресурсов. В большинстве случаев все задачи одного этапа должны завершиться до начала следующего. Процесс распределения указанных задач выполняется объектом `TaskScheduler` и может различаться в зависимости от того, используется ли недискриминационный планировщик или FIFO-планировщик (см. обсуждение в пункте «Планировщик Spark по умолчанию» на с. 40).

В какой-то мере простейший способ представить модель выполнения Spark — считать, что задание Spark представляет собой набор преобразований набора RDD, необходимых для вычисления одного итогового результата. Каждый этап соответствует участку работы, который можно выполнить без вовлечения драйвера. Другими словами, вычисление одного этапа можно осуществить без перемещения данных между секциями. В рамках одного этапа задачи — единицы работы, выполняемой для каждой из секций данных.

Резюме

Фреймворк Spark предлагает передовую эффективную модель параллельных вычислений, основанную на отложенном вычислении неизменяемых, распределенных наборов данных, известных под названием RDD. Spark обеспечивает видимость RDD как интерфейса и возможность использования методов наборов RDD без всяких знаний об их внутренней реализации, однако понимание нюансов их устройства способно помочь при написании более производительного кода. В силу умения Spark выполнять задания параллельно, вычислять задания в нескольких узлах и осуществлять отложенное формирование RDD, подобные логические паттерны могут приводить к широкому варьированию производительности, а ошибки способны проявляться в самых неожиданных местах. Следовательно, чтобы писать и отлаживать код Spark, важно понимать структуру модели его выполнения. Более того, с помощью API Spark часто можно получать одни и те же результаты множеством различных способов, и глубокое понимание того, как происходит выполнение вашего кода, позволит оптимизировать его производительность. В этой книге мы сосредоточимся на способах проектирования приложений Spark, благодаря которым возможно минимизировать сетевой трафик, ошибки памяти и цену сбоев.

3 Наборы DataFrame/Dataset и Spark SQL

Модуль Spark SQL и его интерфейсы `DataFrame` и `Dataset` — будущее Spark в смысле производительности. Они обеспечивают более эффективные варианты хранения данных, продвинутый оптимизатор и операции непосредственно над сериализованными данными. Эти компоненты исключительно важны для максимизации производительности Spark (рис. 3.1).

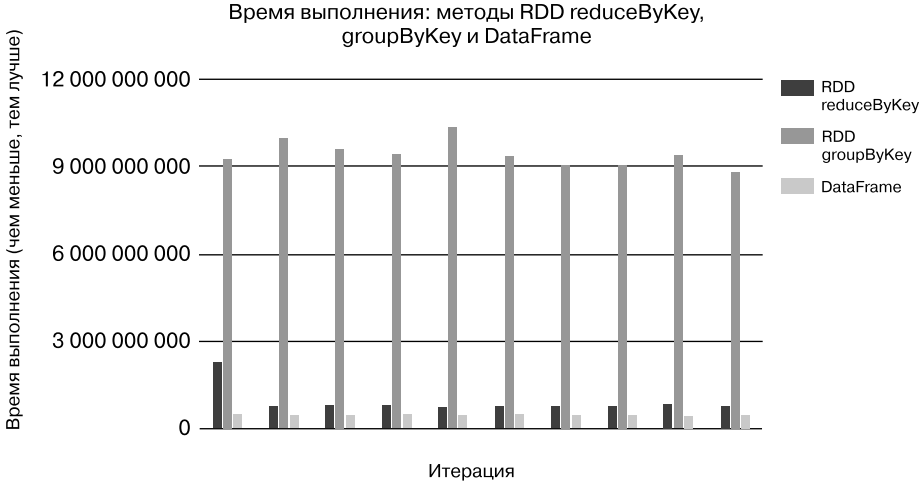

Рис. 3.1. Относительная производительность набора RDD по сравнению с DataFrame, полученная с помощью теста SimplePerfTest, основанного на вычислении средней пушистости панд

Это относительно новые компоненты: интерфейс `Dataset` появился в Spark 1.6, `DataFrame` — в Spark 1.3, а движок SQL — в Spark 1.0. В данной главе мы научим вас

применять инструменты Spark SQL оптимальным образом и сочетать Spark SQL с обычными операциями Spark.

Функциональность наборов DataFrame фреймворка Spark очень отличается от традиционных DataFrame, например, в языках Panda и R. Хотя все они работают со структурированными данными, важно не полагаться на свои наработанные интуитивные представления о DataFrame.

Подобно наборам RDD, наборы `DataFrame` и `Dataset` представляют собой распределенные коллекции с отсутствующей в RDD дополнительной информацией о схеме. Последняя используется для обеспечения более эффективного слоя хранения (tungsten), а также в оптимизаторе (catalyst) для осуществления дополнительных оптимизаций. Помимо информации о схеме, при выполнении операций над наборами `DataFrame` и `Dataset` оптимизатор способен проверять логику, а не просто произвольные функции. Набор `DataFrame` представляет собой набор `Dataset` специальных объектов типа `Row`, у которого отсутствует какая-либо проверка типа на этапе компиляции. Сильно типизированный API Dataset особенно выделяется при использовании с RDD-подобными функциональными операциями. По сравнению с применением наборов RDD при обращении к `DataFrame` оптимизатор Spark может лучше понять наш код и наши данные, благодаря чему становится возможен новый класс оптимизаций, который мы рассмотрим в разделе «Оптимизатор запросов» этой главы.

Несмотря на множество великолепных усовершенствований, вносимых Spark SQL и наборами DataFrame и Dataset, у них все равно есть отдельные «шероховатости» по сравнению с обычной обработкой с помощью «стандартных» наборов RDD. В API Dataset, на момент написания данной книги только появившемся, вероятно, будут внесены некоторые изменения в будущих версиях.

Первые шаги в использовании SparkSession (или HiveContext, или SQLContext)

Подобно тому как класс `SparkContext` является точкой входа для всех приложений Spark, а класс `StreamingContext` — для всех потоковых приложений, класс `SparkSession` служит точкой входа для Spark SQL. Аналогично всем остальным Spark-компонентам для него необходимо импортировать несколько дополнительных компонентов, как показано в примере 3.1.

При использовании командной строки Spark автоматически предоставляется экземпляр класса SparkSession с именем spark в дополнение к объекту SparkContext с именем sc.

Пример 3.1. Импорты Spark SQL

```
import org.apache.spark.sql.{Dataset, DataFrame, SparkSession, Row}
import org.apache.spark.sql.catalyst.expressions.aggregate._
import org.apache.spark.sql.expressions._
import org.apache.spark.sql.functions._
```

Псевдоним типа DataFrame = Dataset[Row] языка Scala в Java не работает — вместо него необходимо использовать Dataset<Row>.

Объект `SparkSession` обычно создается с помощью паттерна «Строитель» (Builder) и метода `getOrCreate()`, возвращающего текущий сеанс, если таковой уже существует. «Строитель» принимает на входе ключи конфигурации в строковом виде `config(key, value)` и сокращения, имеющиеся для множества распространенных параметров. Одно из самых важных сокращений — `enableHiveSupport()`, обеспечивающее доступ к пользовательским функциям UDF СУБД Hive *без* необходимости установить Hive (хотя некоторые дополнительные JAR-файлы все же понадобятся, как вы можете узнать из раздела «Зависимости Spark SQL» далее). Пример 3.2 демонстрирует создание объекта `SparkSession` с поддержкой Hive. Сокращение `enableHiveSupport()` не только настраивает Spark SQL для применения этих Hive JAR, но и проверяет возможность их загрузки, обеспечивая более понятные сообщения об ошибках по сравнению с установкой параметров конфигурации вручную. В целом рекомендуется использовать перечисленные в документации API (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.package) сокращенные формы записи при их наличии, поскольку в универсальном интерфейсе `config` никаких проверок не производится.

Пример 3.2. Создание объекта SparkSession

```
val session = SparkSession.builder()
  .enableHiveSupport()
  .getOrCreate()
// Импортируем implicits[1], в отличие от Spark Core, где неявные функции
// заданы в контексте
import session.implicits._
```

Если при использовании getOrCreate() окажется, что сеанс уже существует, то все ваши параметры конфигурации могут быть проигнорированы и просто будет возвращен этот существующий сеанс. Некоторые параметры, например master, также задействуются только в отсутствие выполняющегося SparkContext, иначе станет применяться этот объект.

[1] Неявные функции, предназначенные для преобразования распространенных объектов языка Scala в наборы Dataset. — *Примеч. пер.*

До выпуска Spark 2.0 вместо `SparkSession` использовались две отдельные точки входа для Spark SQL (`HiveContext` и `SQLContext`). Их названия, вероятно, несколько вводят в заблуждение, так что важно отметить: `HiveContext` *не требует* установки Hive. Основная причина применения `SQLContext` — наличие конфликтов с зависимостями Hive, которые не удается разрешить. В классе `HiveContext` имеется более совершенный синтаксический анализатор по сравнению с классом `SQLContext`, а также дополнительные пользовательские функции (UDF)[1]. Пример 3.3 демонстрирует создание унаследованного от прежних версий `HiveContext`. Лучше всего задействовать класс `SparkSession` по возможности, далее по степени предпочтительности идет `HiveContext`, а затем `SQLContext`. Далеко не все библиотеки, даже не весь код Spark, был модернизирован для использования класса `SparkSession`, и в ряде случаев вы столкнетесь с функциями, которые ожидают на входе объект `SQLContext` или `HiveContext`.

Пример 3.3. Создание экземпляра класса HiveContext

```
val hiveContext = new HiveContext(sc)
// Импортируем implicits;
// в отличие от Spark Core, где неявные функции заданы в контексте
import hiveContext.implicits._
```

При необходимости использовать один из устаревших интерфейсов (`SQLContext` или `HiveContext`) могут оказаться полезны дополнительные импорты, приведенные в примере 3.4.

Пример 3.4. Импорты унаследованных классов Spark SQL

```
import org.apache.spark.sql.SQLContext
import org.apache.spark.sql.hive.HiveContext
import org.apache.spark.sql.hive.thriftserver._
```

Получение объектов SQLContext или HiveContext из объекта SparkSession вне области видимости org.apache.spark поддерживается плохо, однако можно воспользоваться методом getOrCreate.

Зависимости Spark SQL

Подобно остальным компонентам фреймворка Spark, использование Spark SQL требует добавления дополнительных зависимостей. При наличии конфликтов с какими-либо Hive JAR, которые не получается исправить с помощью плагина `shade`, вы оказываетесь ограничены одним только JAR `spark-sql` — хотя хотелось бы

[1] UDF позволяют расширять возможности SQL, например вычислять геопространственное расстояние между точками.

иметь доступ к зависимостям Hive, не прибегая к необходимости включать JAR `spark-hive`.

Для активизации поддержки Hive в объекте `SparkSession` или использования `HiveContext` следует добавить в список зависимостей оба компонента `SQL` и `Hive` фреймворка Spark.

В случае совместимых с Maven систем сборки адреса компонентов `SQL` и `Hive` фреймворка Spark будут следующими: `org.apache.spark:spark-sql_2.11:2.0.0` и `org.apache.spark:spark-hive_2.11:2.0.0`. Пример 3.5 демонстрирует добавление их в «обычную» sbt-сборку, а пример 3.6 — аналогичный процесс для пользователей Maven.

Пример 3.5. Добавление компонентов SQL и Hive фреймворка Spark в «обычную» sbt-сборку

```
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-sql" % "2.0.0",
  "org.apache.spark" %% "spark-hive" % "2.0.0")
```

Пример 3.6. Добавление компонентов SQL и Hive фреймворка Spark в pom-файл Maven

```
<dependency> <!-- Зависимость Spark -->
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.0.0</version>
</dependency>
<dependency> <!-- Зависимость Spark -->
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-hive_2.11</artifactId>
  <version>2.0.0</version>
</dependency>
```

Управление зависимостями Spark

Хотя управлять этими зависимостями вручную — не особенно сложная задача, она чревата ошибками при обновлении версий. Плагин sbt-spark-package (https://github.com/databricks/sbt-spark-package) упрощает управление зависимостями Spark. Обычно он используется для пакетов, созданных сообществом разработчиков (обсуждаемых в пункте «Создание пакета Spark» на с. 299), но полезен и при сборке зависящего от Spark ПО. Для добавления этого плагина в sbt-сборку необходимо создать файл `project/plugins.sbt` и убедиться, что он содержит приведенный в примере 3.7 код.

Пример 3.7. Включение sbt-spark-package в файл project/plugins.sbt

```
resolvers += ["Основной репозиторий пакетов Spark" at
  "https://dl.bintray.com/spark-packages/maven"]

addSbtPlugin("org.spark-packages" % "sbt-spark-package" % "0.2.5")
```

Для функционирования `spark-packages` необходимо указать версию Spark и по крайней мере один компонент Spark (`core`). Это можно сделать в настройках sbt, как показано в примере 3.8.

Пример 3.8. Задание версии Spark и компонента core

```
sparkVersion := "2.1.0"
sparkComponents ++= Seq("core")
```

После установки и настройки плагина `sbt-spark-package` можно добавить требуемые компоненты Spark просто путем добавления `SQL` и `Hive` в список `sparkComponents`, как показано в примере 3.9.

Пример 3.9. Добавление компонентов SQL и Hive фреймворка Spark в сборку sbt-spark-package

```
sparkComponents ++= Seq("sql", "hive", "hive-thriftserver", "hive-thriftserver")
```

Хотя это не обязательно, но если у вас уже есть Hive Metastore, к которому вы хотели бы подключаться с помощью Spark, то можете скопировать файл `hive-site.xml` в каталог `conf/` фреймворка.

Версия Hive Metastore по умолчанию — 1.2.1. Для использования других версий необходимо задать значение свойства spark.sql.hive.metastore.version равным желаемой версии, а также сделать значение spark.sql.hive.metastore.jars равным или maven (чтобы Spark находил и загружал JAR-архивы), или системному пути, по которому находятся JAR-архивы Hive.

Исключение JAR-архивов Hive

Если вы не хотите добавлять в свое приложение зависимости Hive, то можете исключить компонент Hive фреймворка Spark, создав вместо него `SQLContext`, как показано в примере 3.10. Это обеспечивает практически такую же функциональность, но при этом задействуется синтаксический анализатор с меньшими возможностями и отсутствуют некоторые основанные на Hive пользовательские функции (UDF), а также пользовательские функции агрегирования (UDAF).

Пример 3.10. Создание SQLContext

```
val sqlContext = new SQLContext(sc)
// Импортируем implicits;
// в отличие от Spark Core, где неявные функции заданы в контексте
import sqlContext.implicits._
```

Как и базовые объекты `SparkContext` и `StreamingContext`, объект `Hive/SQLContext` используется для загрузки данных. JSON — очень популярный формат, в частности, благодаря удобству его загрузки на различных языках программирования

и по крайней мере частичной удобочитаемости для человека. Некоторые примеры данных из нашей книги именно поэтому и приведены в указанном формате. JSON особенно интересен отсутствием информации о схеме, и фреймворку Spark приходится проделывать некий объем работы для определения схемы по имеющимся данным. Синтаксический разбор JSON может потребовать значительных вычислительных затрат; в ряде простых случаев затраты на синтаксический разбор входных данных в формате JSON превышают затраты на выполнение самой операции.

Мы рассмотрим API для полной загрузки и сохранения JSON в пункте «JSON» на с. 72, но сейчас для начала загрузим образец, подходящий для изучения схемы (пример 3.11).

Пример 3.11. Загрузка данных в формате JSON

```
val df1 = session.read.json(path)
```

Не стесняйтесь загрузить собственные данные в формате JSON, но при отсутствии таковых загляните в примеры из каталога ресурсов на GitHub (https://github.com/high-performance-spark/high-performance-spark-examples/tree/master/resources). Теперь, после загрузки данных в формате JSON, можно приступить к выяснению того, какую схему Spark сумел логически вывести для наших данных.

Основные сведения о схемах

Информация о схеме и об оптимизациях, которые она делает возможными, — одно из основных различий между Spark SQL и базовым Spark. Изучение схемы особенно полезно для `DataFrames`, поскольку в этой ситуации речь не идет о шаблонизированном типе, как в случае наборов RDD или `Dataset`. Spark SQL обычно обрабатывает схемы автоматически, неважно, выведенные логически при загрузке данных или вычисленные на основе информации о родительских `DataFrame` и примененных преобразованиях.

Наборы `DataFrame` предоставляют схему как в удобном для восприятия человеком формате, так и в программном. Метод `printSchema()` служит для отображения схемы набора `DataFrame` и чаще всего используется при работе в командной оболочке, чтобы определить, с чем приходится иметь дело. Он особенно удобен для таких форматов данных, как JSON, в которых схема не понятна сразу после просмотра нескольких записей или чтения заголовка. Получить схему для применения в программе можно просто с помощью вызова метода `schema`, часто используемого в конвейерах преобразований ML. Поскольку вы, вероятно, уже знакомы с case-классами и JSON, то посмотрим, как выглядели бы схемы, эквивалентные примерам 3.12 и 3.13.

Пример 3.12. Данные в формате JSON для схемы

```
{"name":"mission","pandas":[{"id":1,"zip":"94110","pt":"большая", "happy":true,
"attributes":[0.4,0.5]}]}
```

Пример 3.13. Эквивалентный case-класс

```
case class RawPanda(id: Long, zip: String, pt: String,
                    happy: Boolean, attributes: Array[Double])
case class PandaPlace(name: String, pandas: Array[RawPanda])
```

Теперь, описав case-классы, мы можем создать локальный экземпляр, преобразовать его в набор `Dataset` и вывести, как показано в примере 3.14, схему, получив в результате пример 3.15. То же самое можно проделать с данными в формате JSON, но это потребует отдельных настроек, которые мы обсудим в пункте «JSON» на с. 72.

Пример 3.14. Создание набора Dataset на основе case-класса

```
def createAndPrintSchema() = {
  val damao = RawPanda(1, "M1B 5K7", "большая", true, Array(0.1, 0.1))
  val pandaPlace = PandaPlace("toronto", Array(damao))
  val df = session.createDataFrame(Seq(pandaPlace))
  df.printSchema()
}
```

Пример 3.15. Информация о схеме для вложенной структуры (.printSchema())

```
root
 |-- name: string (nullable = true)
 |-- pandas: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- id: long (nullable = false)
 |    |    |-- zip: string (nullable = true)
 |    |    |-- pt: string (nullable = true)
 |    |    |-- happy: boolean (nullable = false)
 |    |    |-- attributes: array (nullable = true)
 |    |    |    |-- element: double (containsNull = false)
```

Помимо схемы в удобном для восприятия человеком формате, информация о схеме доступна также и в виде, подходящем для использования в программе. Эта схема возвращается в качестве `StructField`, как показано в примере 3.16.

Пример 3.16. Case-класс StructField

```
case class StructField(
    name: String,
    dataType: DataType,
    nullable: Boolean = true,
    metadata: Metadata = Metadata.empty)
...
```

В примере 3.17 показана та же схема, что и в примере 3.15, но в пригодном для ввода в программу формате.

Пример 3.17. Информация о схеме для вложенной структуры (.schema()) — отформатировано вручную

```
org.apache.spark.sql.types.StructType = StructType(
  StructField(name,StringType,true),
  StructField(pandas,
    ArrayType(
      StructType(StructField(id,LongType,false),
                 StructField(zip,StringType,true),
                 StructField(pt,StringType,true),
                 StructField(happy,BooleanType,false),
                 StructField(attributes,ArrayType(DoubleType,false),true)),
               true),true))
```

Теперь можно разбираться, что означает эта информация о схеме и как формировать более сложные схемы. Начнем с объекта типа `StructType`, содержащего список полей. Важно отметить: объекты `StructType` могут быть вложенными друг в друга, подобно тому как case-класс может содержать другие case-классы. Поля в `StructType` описываются с помощью `StructField`, в котором задаются имя, тип (см. перечень типов в табл. 3.1 и 3.2) и булево значение, указывающее, может ли значение поля быть неопределенным или отсутствовать.

Таблица 3.1. Основные типы Spark SQL

Тип языка Scala	Тип SQL	Подробности
Byte	ByteType	1-байтные целые числа со знаком (–128,127)
Short	ShortType	2-байтные целые числа со знаком (–32768,32767)
Int	IntegerType	4-байтные целые числа со знаком (–2147483648, 2147483647)
Long	LongType	8-байтные целые числа со знаком (–9223372036854775808, 9223372036854775807)
java.math.BigDecimal	DecimalType	Десятичные дроби с произвольной точностью
Float	FloatType	4-байтное число с плавающей точкой
Double	DoubleType	8-байтное число с плавающей точкой
Array[Byte]	BinaryType	Массив байтов
Boolean	BooleanType	true/false
java.sql.Date	DateType	Дата без информации о времени
java.sql.Timestamp	TimestampType	Дата с информацией о времени (с точностью до секунд)
String	StringType	Символьные строки (хранятся в кодировке UTF8)

Таблица 3.2. Составные типы Spark SQL

Тип языка Scala	Тип SQL	Подробности	Пример
Array[T]	ArrayType (elementType, containsNull)	Массив элементов одного типа, containsNull равен null при наличии элементов со значением null	Array[Int] => ArrayType(IntegerType, true)
Map[K, V]	MapType (elementType,valueType, valueContainsNull)	Ассоциативный массив данных типа «ключ — значение», valuecontainsNull равен null при наличии элементов со значением null	Map[String, Int] => MapType(StringType, IntegerType, true)
case-класс	StructType(List[StructFields])	Поименованные поля возможно неоднородного типа, аналогично case-классу или JavaBean	case class Panda(name: String, age: Int) => StructType(List(StructField("name", StringType, true), StructField("age", IntegerType, true)))

Как вы видели в примере 3.17, StructFields и все составные типы Spark SQL допускают вложение.

Теперь, когда вы знаете, как разбирать и при необходимости описывать схемы данных, вы готовы к изучению интерфейсов класса `DataFrame`.

Схемы Spark SQL вычисляются немедленно, в отличие от соответствующих им данных. Если вы при работе в командной оболочке не вполне уверены в том, что делает какое-либо преобразование, то попробуйте его выполнить и выведите схему. См. пример 3.15.

API DataFrame

API DataFrame модуля Spark SQL позволяет работать с наборами DataFrame, не прибегая к регистрации временных таблиц или генерации SQL-выражений. API DataFrame включает как преобразования, так и действия. Выполняемые над наборами DataFrame преобразования по своей природе более реляционные, в то время как API Dataset (рассматриваемый далее) скорее функциональный.

Преобразования

Выполняемые над наборами `DataFrame` преобразования аналогичны по своему характеру преобразованиям RDD, но с более реляционным оттенком. Вместо произвольных функций, недоступных для анализа оптимизатора, используется ограниченный синтаксис выражений, благодаря которому оптимизатор получает больше информации. Как и в случае с наборами RDD, у нас есть широкие возможности разбиения преобразований на простые с одним набором `DataFrame`, несколькими наборами `DataFrame`, данными типа «ключ — значение», а также оконные/группирующие преобразования.

Преобразования Spark SQL являются отложенными только частично, схема вычисляется немедленно.

Простые преобразования наборов DataFrame и SQL выражения

Простые преобразования наборов `DataFrame` позволяют делать практически все, для чего достаточно построчной обработки данных[1]. С их помощью можно совершать большинство выполняемых над наборами RDD операций, за исключением использования выражения Spark SQL вместо произвольных функций. Для иллюстрации этого мы начнем с рассмотрения различных видов операций фильтрации, доступных для наборов `DataFrame`.

Функции `DataFrame`, такие как `filter`, принимают на входе выражения Spark SQL вместо лямбда-выражений. Благодаря им оптимизатор понимает, что представляет собой условие, и в случае `filter` часто может избежать чтения ненужных записей.

Для начала рассмотрим SQL-выражение для выборки из наших данных несчастных панд на основе имеющейся схемы. Первый шаг: поиск столбца с нужной информацией. В нашем случае таковым является столбец `happy`, и для нашего набора `DataFrame` (с именем `df`) можно обратиться к этому столбцу с помощью функции `apply` (скажем, `df("happy")`). По условию `filter` требуется, чтобы возвращалось булево значение, и если бы мы хотели выбрать счастливых панд, то могли бы извлекать значение столбца во всем SQL-выражении целиком. Однако ввиду того, что мы хотим найти несчастных панд, достаточно проверять, не равно ли оно `true`, с помощью оператора `!==`, как показано в примере 3.18.

Пример 3.18. Простой фильтр для отбора несчастных панд

```
pandaInfo.filter(pandaInfo("happy") !== true)
```

[1] Построчная обработка позволяет осуществлять узкие преобразования без перетасовок.

Чтобы найти столбец, можно указать его имя для конкретного DataFrame или воспользоваться неявным оператором $ для поиска столбца. Второй вариант особенно удобен в случае анонимного объекта DataFrame. Функцию побитового отрицания можно применять вместе с $, чтобы упростить выражение из примера 3.18 до df.filter(!$("happy")).

Этот пример иллюстрирует возможность обращения к конкретному столбцу набора `DataFrame`. Для обращения к другим (допустим, вложенным) структурам в `DataFrame`, картам соответствий с ключами и элементам массивов подойдет тот же синтаксис. Так, если первый элемент массива `attributes` соответствует мягкости на ощупь и нас интересуют только очень мягкие на ощупь панды, можно обратиться к данному элементу следующим образом: `df("attributes")(0) >= 0.5`.

Наши выражения не обязаны ограничиваться одним столбцом. Можно сравнивать несколько столбцов в выражении для фильтрации. Сложные фильтры, подобные показанному в примере 3.19, сложнее спустить на уровень хранилища, так что вы вряд ли обнаружите такое же ускорение по сравнению с наборами RDD, как в случае простых фильтров.

Пример 3.19. Более сложный фильтр

```
pandaInfo.filter(
  pandaInfo("happy").and(pandaInfo("attributes")(0) > pandaInfo("attributes")(1))
)
```

Столбцовые операторы модуля Spark SQL определяются для класса столбца, так что фильтр с выражением 0 >= df.col("friends") компилироваться не будет, ведь Scala попытается применить определенный для 0 оператор >=. Вместо этого следует написать df.col("friend") <= 0 или преобразовать 0 в столбцовый литерал с помощью функции lit[1].

API DataFrame модуля Spark SQL содержит огромное множество доступных операторов. Можно использовать все стандартные математические операторы с плавающей точкой, а также стандартные логические и побитовые операторы (в начале названия которых стоит `bitwise`, чтобы отличать их от логических). При операциях над столбцами для равенства/неравенства применяются операторы `===` и `!==` с целью избежать конфликта с внутренними операторами Scala. Для столбцов строк доступны функции `startsWith/endsWith`, `substr`, `like` и `isNull`. Полный список операций приведен на http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column и в табл. 3.3 и 3.4.

[1] Столбцовый литерал — это столбец с фиксированным значением, не меняющийся от строки к строке.

Таблица 3.3. Операторы языка Scala модуля Spark SQL

Оператор Scala	Эквивалент в Java	Тип входного столбца	Выходной тип	Назначение	Пример	Результат
!==	notEqual	Любой	Булев	Проверяет выражения на неравенство	"привет" !== "пока"	true
%	mod	Числовой	Числовой	Остаток при целочисленном делении	10 % 5	0
&&	and	Булев	Булев	Логическое И	true && false	false
*	multiply	Числовой	Числовой	Умножение	2 * 21	42
+	plus	Числовой	Числовой	Сложение	2 + 2	4
–	minus	Числовой	Числовой	Вычитание	2 – 2	0
–	unary_	Числовой	Числовой	Изменение знака	–42	–42
/	division	Числовой	Число типа double	Деление	43 / 2	21.5
<	lt	Сравнимый	Булев	Меньше	"a" < "b"	true
<=	leq	Сравнимый	Булев	Меньше или равно	"a" <= "a"	true
===	equals	Любой	Любой	Проверка на равенство (небезопасно в случае неопределенных значений)	"a" === "a"	true
<=>	eqNullSafe	Любой	Любой	Проверка на равенство (можно безопасно использовать в случае неопределенных значений)	"a" <=> "a"	true
>	gt	Сравнимый	Булев	Больше	"a" > "b"	false
>=	geq	Сравнимый	Булев	Больше или равно	"a" >= "b"	false

Таблица 3.4. Операторы выражений Spark SQL

Оператор Scala	Типы входного столбца	Выходной тип	Назначение	Пример	Результат
apply	Составной тип	Тип поля, к которому обращаются	Получение значения из объекта составного типа (например, просмотр structfield/карты или индекса массива)	[1,2,3].apply(0)	1
bitwiseAND	Целочисленный тип[1]	Аналогично входному типу	Вычисление побитового И	21.bitwiseAND(11)	1
bitwiseOR	Целочисленный тип	Аналогично входному типу	Вычисление побитового ИЛИ	21.bitwiseOR(11)	31
bitwiseXOR	Целочисленный тип	Аналогично входному типу	Вычисление побитового исключающего ИЛИ	21.bitwiseXOR(11)	30

Не все выражения Spark SQL можно использовать в любом вызове API. Например, соединения Spark SQL не поддерживают составные операции, а filter требует, чтобы результат выражения был булевым, и т. д.

Помимо непосредственно применяемых к столбцам операторов, на https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$ можно найти еще больший набор функций для использования со столбцами, часть из которых приведена в табл. 3.5–3.7. В качестве примера показаны значения для каждого столбца в конкретной строке, но не забывайте, что эти функции вызываются для столбцов, а не значений.

Таблица 3.5. Стандартные функции Spark SQL

Название функции	Назначение	Входные типы	Пример использования	Результат
lit(value)	Преобразование символьного значения языка Scala в столбцовый литерал	Столбец и символьное значение	lit(1)	Column(1)
array	Создание нового столбца массива	Должны все относиться к одному типу Spark SQL	array(lit(1),lit(2))	array(1,2)
isNaN	Проверка на значение NaN	Числовой	isNan(lit(100.0))	false
not	Обратное значение	Булев	not(lit(true))	false

[1] К целочисленным типам относятся ByteType, IntegerType, LongType и ShortType.

Таблица 3.6. Часто используемые математические выражения Spark SQL

Название функции	Назначение	Входные типы	Пример использования	Результат
abs	Абсолютное значение	Числовое	abs(lit(–1))	1
sqrt	Квадратный корень	Числовое	sqrt(lit(4))	2
acos	Арккосинус	Числовое	acos(lit(0.5))	1.04...[1]
asin	Арксинус	Числовое	asin(lit(0.5))	0.523...
atan	Арктангенс	Числовое	atan(lit(0.5))	0.46...
cbrt	Кубический корень	Числовое	sqrt(lit(8))	2
ceil	Округление в большую сторону	Числовое	ceil(lit(8.5))	9
cos	Косинус	Числовое	cos(lit(0.5))	0.877...
sin	Синус	Числовое	sin(lit(0.5))	0.479...
tan	Тангенс	Числовое	tan(lit(0.5))	0.546...
exp	Экспонента	Числовое	exp(lit(1.0))	2.718...
floor	Округление в меньшую сторону	Числовое	floor(lit(8.5))	8
least	Минимальное из значений	Числовые	least(lit(1), lit(–10))	–10

Таблица 3.7. Функции, используемые с массивами Spark SQL

Название функции	Назначение	Пример использования	Результат
array_contains	Проверка, содержит ли массив заданное значение	array_contains (lit(Array(2,3,–1), 3))	true
sort_array	Сортировка массива (по умолчанию в порядке возрастания)	sort_ array(lit(Array(2,3,–1)))	Array(–1,2,3)
explode	Создание строки для каждого элемента массива. Часто бывает полезно при работе с вложенными записями в формате JSON. Принимает на входе или название столбца, или дополнительную функцию, сопоставляющую строки с итераторами case-классов	explode(lit(Array(2,3,–1)), "murh")	Row(2),Row(3), Row(–1)

Помимо простой фильтрации данных, можно также создать объект `DataFrame` с новыми столбцами или обновленными значениями в старых. Фреймворк Spark задействует для этого синтаксис выражений, который мы обсуждали для `filter`, за исключением того, что вместо необходимости применять условие (например, проверку на равенство) результаты используются в качестве значений нового объекта `DataFrame`. Чтобы продемонстрировать применение `select` для составных

[1] Здесь и далее сокращено для удобства отображения.

и обычных типов данных, в примере 3.20 фигурирует функция `explode` Spark SQL для преобразования входного объекта `DataFrame` со значениями типа `PandaPlace` в объект `DataFrame` из значений просто типа `PandaInfo`, а также вычисления показателя «мягкость на ощупь» каждой из панд.

Пример 3.20. Операторы select и explode модуля Spark SQL

```
val pandaInfo = pandaPlace.explode(pandaPlace("pandas")){
  case Row(pandas: Seq[Row]) =>
    pandas.map{
      case (Row(
        id: Long,
        zip: String,
        pt: String,
        happy: Boolean,
        attrs: Seq[Double])) =>
        RawPanda(id, zip, pt, happy, attrs.toArray)
    }}
pandaInfo.select(
   (pandaInfo("attributes")(0) / pandaInfo("attributes")(1))
    .as("squishyness"))
```

При формировании последовательности операций сгенерированные имена столбцов быстро становятся громоздкими. В этом случае задать имя итогового столбца помогут операторы as и alias.

Несмотря на большие возможности всех указанных операций, иногда встречается логика, которую удобнее выражать с помощью семантики `if/else`. Одним из простых образцов этого может послужить приведенное в примере 3.21 кодирование различных типов панд числовыми значениями[1]. Для создания подобного эффекта можно связать цепочкой функции `when` и `otherwise`.

Пример 3.21. If/else в Spark SQL

```
/**
  * Кодирует pandaType целочисленными значениями вместо строковых
  *
  * @param pandaInfo входной DataFrame
  * @return возвращает DataFrame из pandaId и целочисленных значений для pandaType
  */
def encodePandaType(pandaInfo: DataFrame): DataFrame = {
  pandaInfo.select(pandaInfo("id"),
    (when(pandaInfo("pt") === "большая", 0).
    when(pandaInfo("pt") === "красная", 1).
    otherwise(2)).as("encodedType")
  )
}
```

[1] Функция StringIndexer (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.feature.StringIndexer) из конвейера ML была специально создана для кодирования строковых индексов.

Специализированные преобразования DataFrame для отсутствующих и зашумленных данных

Модуль Spark SQL также предоставляет специальные инструменты для работы с отсутствующими, неопределенными и некорректными данными. Благодаря использованию функций `isNan` и `isNull` совместно с фильтрами можно формировать условия для строк, которые надо оставить. Например, при наличии некоего количества различных столбцов, вероятно, с различными уровнями точности (причем часть из них могут быть неопределенными) можно задействовать `coalesce(c1, c2, ..)`, которая вернет первый, не равный `null` столбец. Аналогично для числовых данных `nanvl` возвращает первое не равное `NaN` значение (например, `nanvl(0/0, sqrt(-2), 3)` возвращает `3`). Чтобы упростить взаимодействие с отсутствующими данными, функция `na` для объектов `DataFrame` позволяет обращаться к некоторым утилитам из класса `DataFrameNaFunctions` (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameNaFunctions), часто используемым для работы с отсутствующими данными.

Не только построчные преобразования

Иногда принятия решения построчно, доступного с помощью `filter`, недостаточно. Модуль Spark SQL позволяет сделать выборку неповторяющихся строк путем вызова метода `dropDuplicates`, но, как и в случае с аналогичной операцией над наборами RDD (`distinct`), это требует перетасовки, поэтому зачастую выполняется намного медленнее, чем `filter`. В отличие от RDD, `dropDuplicates` может дополнительно отбрасывать строки на основе лишь подмножества столбцов, таких как поля ID, что и показано в примере 3.22.

Пример 3.22. Исключение дублирующихся идентификаторов панд

```
pandas.dropDuplicates(List("id"))
```

Таким образом, мы изящно переходим к следующему пункту, посвященному агрегирующим функциям и функции `groupBy`, поскольку наиболее затратной составляющей их всех является перетасовка.

Агрегирующие функции и функция groupBy

В модуле Spark SQL имеется множество агрегирующих функций, обладающих широкими возможностями, а его оптимизатор позволяет легко объединять несколько таких функций в одно действие/один запрос. Так же как в объектах `DataFrame` библиотеки Pandas, функция `groupBy` возвращает специальные объекты, над которыми можно осуществлять отдельные виды агрегирования. В версиях фреймворка Spark, предшествующих 2.0, для этого использовался универсальный класс `GroupedData` (http://spark.apache.org/docs/1.6.2/api/scala/index.html#org.apache.spark.sql.GroupedData), но

в версии 2.0 и более поздних преобразование `groupBy` для наборов `DataFrame` такое же, как и для `Dataset`.

Возможности агрегирующих функций для наборов `Dataset` расширены, они возвращают `GroupedDataset` (http://spark.apache.org/docs/1.6.2/api/scala/index.html#org.apache.spark.sql.GroupedDataset) (в предшествующих 2.0 версиях фреймворка Spark) или `KeyValueGroupedDataset` (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.KeyValueGroupedDataset) при группировке с произвольной функцией и `RelationalGroupedDataset` (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.RelationalGroupedDataset) при группировке с реляционным/Dataset DSL-выражением. Мы обсудим дополнительные «типизированные» свойства в подразделе «Группирующие операции с наборами Dataset» на с. 86, а обычную «нетипизированную» функциональность `groupBy` для `DataFrame` и `Dataset` рассмотрим тут.

Агрегирующие функции `min`, `max`, `avg` и `sum` реализованы как удобные методы непосредственно класса `GroupedData`, причем можно описать и другие путем передачи выражений методу `agg`. Пример 3.23 показывает, как вычислить максимальный размер панд, относящихся к одному почтовому индексу. Достаточно указать сводные показатели, которые нужно вычислить, и результаты будут возвращены в виде объекта `DataFrame`.

Если вы привыкли использовать наборы RDD, то вас может беспокоить безопасность groupBy, но в настоящий момент это вполне безопасная операция над DataFrame благодаря оптимизатору Spark SQL, который автоматически организует конвейер из сверток, избегая масштабных перетасовок и огромных записей.

Пример 3.23. Вычисление максимального размера панд, относящихся к одному почтовому индексу

```
def maxPandaSizePerZip(pandas: DataFrame): DataFrame = {
  pandas.groupBy(pandas("zip")).max("pandaSize")
}
```

Хотя в примере 3.23 максимум вычисляется по ключам, указанные агрегирующие функции можно применять и ко всему набору `DataFrame`, или ко всем его числовым столбцам. Это часто бывает удобно при сборе сводных статистических показателей для имеющихся данных. На самом деле существует встроенное преобразование `describe`, используемое в примерах 3.24 и 3.25, которое именно это и делает, хотя его можно и ограничить частью столбцов.

Пример 3.24. Вычисление распространенных статистических показателей, включая количество, среднее значение, стандартное отклонение и другие, по всему набору DataFrame

```
// Вычисляем статистические показатели: количество, среднее значение,
// стандартное отклонение, минимальное и максимальное значения
// всех числовых полей имеющейся информации о пандах
```

```
// Нечисловые поля (например, строки (имена) или массивы) игнорируются
val df = pandas.describe()
// С помощью метода collect() отправляем обратно полученные
// локально статистические показатели
println(df.collect())
```

Пример 3.25. Результаты выполнения методов describe и collect для небольшого подмножества данных (примечание: вычисляются сводные показатели всех числовых полей)

```
Array([count,3,3], [mean,1.3333333333333333,5.0],
  [stddev,0.5773502691896258,4.358898943540674], [min,1,2], [max,2,10])
```

Поведение функции groupBy менялось в Spark от версии к версии. До версии 1.3 значения группируемых столбцов по умолчанию исключались из выводимых данных, в то время как после 1.3 они там сохраняются. Существует параметр конфигурации, spark.sql.retainGroupColumns, установка значения которого в false приводит к принудительному использованию предыдущей версии поведения.

Для вычисления нескольких различных сводных показателей или более сложных сводных показателей следует задействовать API `agg` класса `GroupedData` вместо непосредственного вызова методов `count`, `mean` и тому подобных функций. При использовании API `agg` необходимо указать или список выражений агрегирования, описывающих сводные показатели строки, или карту соответствий имен столбцов именам агрегирующих функций. После вызова `agg` с требуемыми сводными показателями будет возвращен обычный набор `DataFrame` с результатами агрегирования. Что касается обычных функций, они перечислены в документации по языку Scala (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$). В табл. 3.8 приведены некоторые из часто применяемых и полезных агрегирующих функций. В примерах из этой таблицы мы будем использовать набор `DataFrame` со схемой, состоящей из полей имени (строка) и возраста (целое число), причем оба поля допускают неопределенное значение, — со значениями (`{"ikea", null}, {"tube", 6}, {"real", 30}`). Пример 3.26 демонстрирует, как вычислить минимальное и среднее значения для столбца `pandaSize` нашего текущего образца с пандами.

Вычислять несколько сводных показателей с помощью Spark SQL гораздо проще, чем с API RDD.

Пример 3.26. Вычисление сводных показателей с применением API agg

```
def minMeanSizePerZip(pandas: DataFrame): DataFrame = {
  // Вычисляем минимальное и среднее значения
  pandas.groupBy(pandas("zip")).agg(
    min(pandas("pandaSize")), mean(pandas("pandaSize")))
}
```

Таблица 3.8. Агрегирующие функции модуля Spark SQL для использования с API agg

Название функции	Назначение	Требования к хранению	Входные типы	Пример использования	Пример результата
approxCount Distinct[1]	Подсчет приблизительного количества различных значений в столбце[2]	Настраивается с помощью параметра rsd (управляющего частотой ошибок)	Все	df.agg(approx CountDistinct (df("age"), 0.001))	2
avg	Среднее значение	Постоянные	Числовые	df.agg(avg(df ("age")))	18
count	Подсчитывает количество элементов (исключая null). В особом случае «*» подсчитывает количество строк	Постоянные	Все	df.agg(count (df("age")))	2
countDistinct	Подсчитывает количество различных значений в столбце	0 (различающиеся элементы)	Все	df.agg(count Distinct(df ("age")))	2
first	Возвращает первый элемент[3]	Постоянные	Все	df.agg(first (df("age")))	6
last	Возвращает последний элемент	Постоянные	Все	df.agg(last(df ("age")))	30
stddev	Выборочное стандартное отклонение	Постоянные	Числовые	df.agg(stddev (df("age")))	16.97...
stddev_pop	Стандартное отклонение[4]	Постоянные	Числовые	df.agg(stddev_ pop(df("age")))	12.0
sum	Сумма значений	Постоянные	Числовые	df.agg(sum(df ("age")))	36
sumDistinct	Сумма различающихся значений	0 (различающиеся элементы)	Числовые	df.agg(sum Distinct (df("age")))	36

Продолжение ⇩

[1] Устарела и исключена, начиная с версии Spark 2.1.0.

[2] Реализовано с помощью алгоритма HyperLogLog: https://en.wikipedia.org/wiki/HyperLogLog.

[3] Эта функция часто использовалась в ранних версиях Spark, когда в выводе не сохранялся столбец, по которому производится группировка.

[4] Добавлено в Spark 1.6.

Таблица 3.8 *(продолжение)*

Название функции	Назначение	Требования к хранению	Входные типы	Пример использования	Пример результата
min	Минимальное значение	Постоянные	Допускающие сортировку данных	df.agg(min(df ("age")))	5
max	Максимальное значение	Постоянные	Допускающие сортировку данных	df.agg(max(df ("age")))	30
mean	Среднее значение	Постоянные	Числовые	df.agg(mean(df ("age")))	18

Помимо использования агрегирующих функций с groupBy, можно применять их же к многомерным кубам (с помощью функции cube) и сводным таблицам (задействуя rollup).

Если встроенных агрегирующих функций недостаточно, то можно расширить свойства Spark SQL с помощью UDF, как обсуждается в разделе «Расширение классов пользовательскими функциями, а также пользовательскими функциями агрегирования (UDF, UDAF)» далее в текущей главе, хотя это, вероятно, не так просто.

Оконные функции

В Spark SQL 1.4.0 появились оконные функции, упрощающие работу с диапазонами или окнами строк. При создании окна необходимо указать, на какие строки оно распространяется, порядок строк в пределах каждой секции/группы и размер окна (допустим, K строк до и J строк после ИЛИ-диапазона между значениями). Для большей наглядности на рис. 3.2 показан пример организации окон с ее результатами. При таком описании каждая входная строка оказывается соотнесена с каким-либо набором строк, называемым кадром (frame), который используется для вычисления итогового результата. Оконные функции очень удобны для вычисления, скажем, средней скорости в случае зашумленных данных, относительных продаж и т. п. Окно для отображения возраста панд показано в примере 3.27.

Пример 3.27. Описание окна для отображения ±10 ближайших по возрасту панд, отсортированных по почтовому индексу

```
  val windowSpec = Window
    .orderBy(pandas("age"))
    .partitionBy(pandas("zip"))
    .rowsBetween(start = -10, end = 10)
// Вместо этого можно воспользоваться для задания диапазона
// методом rangeBetween
```

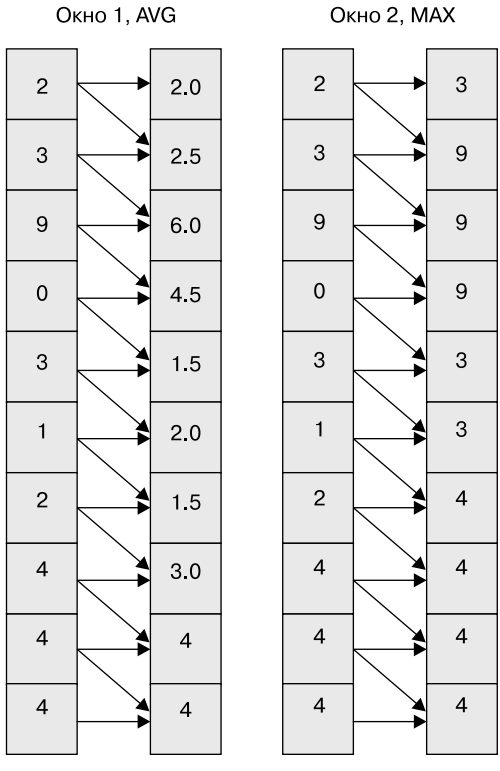

Рис. 3.2. Организация окон в Spark SQL

После определения характеристик окна можно вычислить какую-либо функцию по нему, как показано в примере 3.28. Существующие агрегирующие функции фреймворка Spark, описанные в пункте «Агрегирующие функции и функция groupBy» на с. 62, дают возможность рассчитать сводные показатели по окнам. Оконные операции очень удобны для таких операций, как калмановская фильтрация или различные виды относительного анализа.

Пример 3.28. Вычисление отклонения от среднего значения с помощью окна для отображения ±10 ближайших по возрасту панд, отсортированных по почтовому индексу

```
val pandaRelativeSizeCol = pandas("pandaSize") -
  avg(pandas("pandaSize")).over(windowSpec)

pandas.select(pandas("name"), pandas("zip"), pandas("pandaSize"),
pandas("age"), pandaRelativeSizeCol.as("panda_relative_size"))
```

На момент написания данной книги оконные функции требуют включения поддержки Hive или использования объекта HiveContext.

Сортировка

Сортировка допустима по нескольким столбцам в возрастающем (по умолчанию) или убывающем порядке. Эти порядки можно смешивать, как показано в примере 3.29. У Spark SQL есть дополнительные преимущества при сортировке, поскольку определенные сериализованные данные можно сравнивать и без десериализации.

Пример 3.29. Сортировка по возрасту и размеру панд в противоположных порядках

```
pandas.orderBy(pandas("pandaSize").asc, pandas("age").desc)
```

Сортировка часто оказывается полезна для ограничения результатов — возврата только верхних или нижних K результатов. Задать количество строк при ограничении можно с помощью метода `limit(numRows)`, ограничивающего количество строк в наборе `DataFrame`. Ограничения также иногда используются для отладки без сортировки, чтобы возвращался лишь небольшой результат. Если же вместо ограничения количества строк в соответствии с порядком сортировки нужно произвести выборку данных, то загляните в одноименный подраздел на с. 236, в котором описаны методы выполнения выборки в Spark SQL.

Преобразования нескольких наборов DataFrame

Помимо выполнения преобразований над одним набором `DataFrame`, можно выполнять операции и над несколькими такими наборами. В голову прежде всего приходят, вероятно, различные типы соединений, рассмотренные в главе 4, но, кроме них, над объектами `DataFrame` можно также совершать немало операций наподобие операций с множествами.

Операции с множествами. Различные операции с `DataFrame` позволяют выполнять многое из того, что обычно рассматривается как операции с множествами. Они несколько отличаются от традиционных операций с множествами, поскольку отсутствует ограничение на уникальность элементов. Хотя вы, наверное, уже знакомы с результатами подобных операций из обычного Spark и книги *Learning Spark*, мы приведем данные о вычислительных затратах на эти операции в табл. 3.9.

Таблица 3.9. Операции с множествами

Название операции	Затраты
unionAll	Низкие
intersect	Высокие
except	Высокие
distinct	Высокие

Простые SQL-запросы в старом стиле и взаимодействие с данными из Hive

Иногда удобнее применять обычные SQL-запросы вместо того, чтобы совершать операции с наборами `DataFrame`. При наличии подключения к хранилищу Hive Metastore можно писать SQL-запросы непосредственно к таблицам Hive и получать результаты в виде наборов `DataFrame`. Если вы хотите писать запросы к конкретному набору `DataFrame`, то можно зарегистрировать его в качестве временной таблицы, как показано в примере 3.30 (или сохранить его в качестве управляемой таблицы при условии, что планируете использовать его в нескольких заданиях). Объекты `Dataset` можно также преобразовывать обратно в объекты `DataFrame` и регистрировать для выполнения к ним запросов:

Пример 3.30. Регистрация/сохранение таблиц

```
def registerTable(df: DataFrame): Unit = {
  df.registerTempTable("pandas")
  df.write.saveAsTable("perm_pandas")
}
```

Запросы остаются такими же, независимо от того, временная ли это таблица, существующая таблица Hive или только что сохраненная таблица Spark, как показано в примере 3.31.

Пример 3.31. Выполнение запроса к таблице (временной или постоянной)

```
def querySQL(): DataFrame = {
  sqlContext.sql("SELECT * FROM pandas WHERE size > 0")
}
```

Можно также писать запросы и без регистрации таблиц, указывая конкретный путь к файлу, как показано в примере 3.32.

Пример 3.32. Выполнение запроса к исходному файлу

```
def queryRawFile(): DataFrame = {
  sqlContext.sql("SELECT * FROM parquet.`path_to_parquet_file`")
}
```

Представление данных в наборах DataFrames и Datasets

Наборы `DataFrame` — не просто RDD-коллекции объектов типа `Row`, они обладают специализированным представлением и столбцовым форматом кэша. Данное представление не только более экономично в смысле занимаемого места, но и кодируется намного быстрее, чем сериализация Kryo. Точнее говоря, как и наборы RDD,

наборы `DataFrame` и `Dataset` обычно вычисляются отложенным образом с формированием «родословной» зависимостей (не считая того, что в `DataFrame` это носит название логического плана и содержит больший объем информации).

Tungsten. Это новый компонент Spark SQL, обеспечивающий более эффективное выполнение операций Spark за счет работы непосредственно на уровне байтов. Взглянем еще раз на рис. 3.1 и детальнее рассмотрим вопрос различий в требуемом для хранения пространстве между наборами RDD и `DataFrame` при кэшировании (рис. 3.3). Компонент Tungsten включает специализированные структуры, предназначенные для хранения данных в оперативной памяти и специально приспособленные для операций, требующихся фреймворку Spark. Он также содержит усовершенствованную генерацию кода и специализированный проводной протокол.

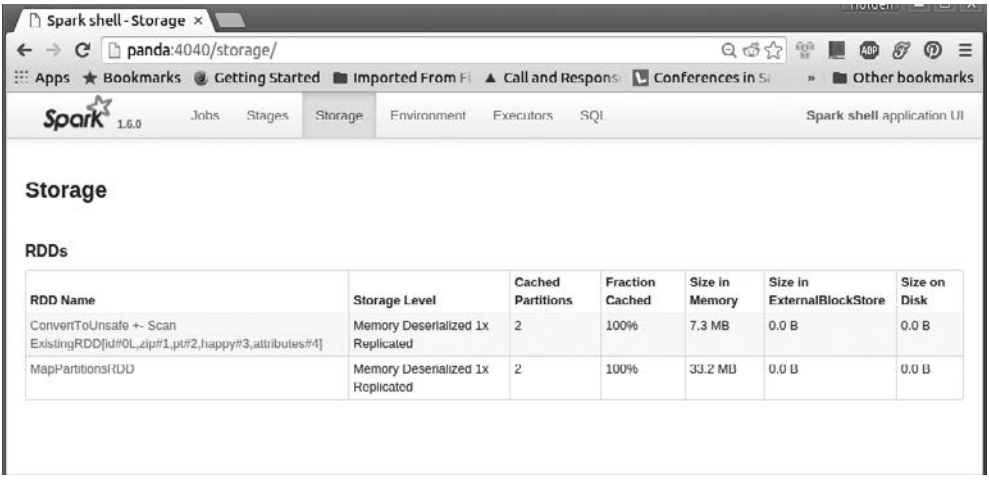

Рис. 3.3. Занимаемое одними и теми же данными место при хранении набора RDD по сравнению с набором DataFrame

Если вы ранее работали с Hadoop, то можете считать типы данных компонента Tungsten типами WritableComparable «на стероидах».

Tungsten-представление гораздо компактнее объектов, сериализованных с помощью Java или даже Kryo. Tungsten не зависит от объектов языка Java, поэтому поддерживает выделение памяти как в куче, так и вне ее. Не только формат оказывается более компактным, но и сериализация занимает намного меньше времени, чем нативная.

Поскольку Tungsten не зависит от работы с объектами языка Java, можно хранить данные как в куче (в JVM), так и вне ее. В последнем случае важно оставлять достаточно места в контейнерах для подобных выделений памяти, приблизительно оценить которые позволяет веб-интерфейс.

Структуры данных Tungsten были созданы с учетом специфики обработки данных, в которой они используются. Классический пример — сортировка, распространенная и весьма ресурсоемкая операция. Представление для проводного протокола реализовано так, что сортировку можно выполнять без повторной десериализации данных.

В будущем Tungsten может привести к повышению вероятности использования некоторых не-JVM-библиотек. Для многих простых операций большую часть затрат на применение из JVM BLAS или других подобных пакетов для линейной алгебры составляет стоимость копирования данных вне кучи.

Благодаря отсутствию расходов на оперативную память и сборку мусора вследствие использования обычных Java-объектов Tungsten способен обрабатывать бóльшие наборы данных, чем аналогичные созданные вручную агрегирующие функции. Tungsten по умолчанию встроен в Spark 1.5 и может быть активизирован в предыдущих версиях путем присвоения параметру `spark.sql.tungsten.enabled` значения `true` (или отключен в последующих версиях путем установки этого параметра в `false`). Даже без Tungsten Spark SQL задействует столбцовый формат хранения с сериализацией Kryo для минимизации затрат на хранение.

Функции загрузки и сохранения данных

Способ загрузки и сохранения данных модуля Spark SQL отличается от используемого в базовом Spark. Для спуска определенных типов операций на уровень хранилища в Spark SQL существует собственный API источников данных (Data Source API) (https://databricks.com/blog/2015/01/09/spark-sql-data-sources-api-unified-data-access-for-the-spark-platform.html). Источники данных позволяют управлять тем, какие типы операций спускаются на их уровень. Разработчику не нужно задумываться о происходящих при этом внутренних действиях, за исключением случая, когда необходимый ему источник данных не поддерживается.

Загрузка данных в Spark SQL не настолько отложена, как в обычном Spark, но в целом выполняется отложенным образом. Проверить это можно, попытавшись загрузить данные из несуществующего источника.

DataFrameWriter и DataFrameReader

Интерфейсы DataFrameWriter (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameWriter) и DataFrameReader (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameReader) обеспечивают чтение данных из внешних источников. Обратиться к первому из них можно, вызвав метод `write` объектов `DataFrame` или `Dataset`, ко второму — вызвав метод `read` объекта `SQLContext`.

Spark SQL обновил API загрузки/сохранения в Spark 1.4, так что может встретиться код, по-прежнему использующий старый вариант API без классов чтения/записи наборов DataFrame, но, так сказать, за кадром он реализован в виде адаптера для нового API.

Форматы

Задать формат при чтении или записи можно, вызвав метод `format(formatName)` классов `DataFrameWriter/DataFrameReader`. Отражающие специфику конкретного формата параметры, например количество выбираемых в виде JSON записей, задаются или при передаче карты параметров `options`, или при установке с помощью `options` отдельных параметров в читающем/записывающем классе.

Методы для форматов, поддерживаемых Apache, — JSON, JDBC, ORC и Parquet — задаются непосредственно в объектах чтения/записи с помощью передачи пути или информации о соединении. Эти методы призваны сделать работу более удобной и представляют собой адаптеры для более общих методов, которые мы продемонстрируем в текущей главе.

JSON

Загрузка и запись данных в формате JSON поддерживается непосредственно модулем Spark SQL, и, несмотря на отсутствие в JSON информации о схеме, Spark SQL способен выводить логически схему с помощью выборки записей. Загрузка данных в формате JSON требует больших затрат, чем загрузка многих источников данных, поскольку фреймворку Spark приходится читать часть записей, чтобы выяснить информацию о схеме. Если последняя сильно отличается от записи к записи (или количество записей очень невелико), то можно повысить процент читаемых для определения схемы записей, скажем увеличив значения параметра `samplingRatio`, как в примере 3.33, где мы задаем его равным `100%`.

Пример 3.33. Загрузка данных в формате JSON с использованием всех (100 %) записей для определения схемы

```
val df2 = session.read.format("json")
  .option("samplingRatio", "1.0").load(path)
```

Механизм определения схемы Spark — веский довод в пользу того, чтобы применять фреймворк для работы с данными в формате JSON, даже если размер данных позволяет обработать их в отдельном узле.

Наши входные данные могут содержать некорректные записи в формате JSON, которые желательно отфильтровать, так что не помешает получать на входе и RDD-набор строк. Это позволит загрузить входные данные в виде обычного текстового файла, отфильтровать некорректные записи, после чего загрузить данные в JSON. Выполнить указанные операции можно с помощью встроенной функции `json` класса `DataFrameReader`, принимающей на входе наборы RDD или пути и показанной в примере 3.34. Методы преобразования типа наборов обычных объектов мы рассмотрим в пункте «Наборы RDD» на с. 77.

Пример 3.34. Загрузка jsonRDD

```
val rdd: RDD[String] = input.filter(_.contains("panda"))
val df = session.read.json(rdd)
```

JDBC

Источник данных JDBC представляет собой естественный источник данных Spark SQL, поддерживающий множество операций. Поскольку реализации JDBC у разных поставщиков СУБД несколько различаются, то понадобится добавить JAR-архив для вашего конкретного JDBC-источника данных. А ввиду того, что типы полей SQL тоже различаются, фреймворк Spark использует класс `JdbcDialects`, в который встроены «диалекты» для `DB2`, `Derby`, `MsSQL`, `MySQL`, `Oracle` и `Postgres`[1].

Хотя фреймворк Spark поддерживает множество различных JDBC-источников, JAR-файлы, необходимые для взаимодействия со всеми этими базами данных, не включены в него. При инициализации выполнения Spark-задания с помощью команды `spark-submit` требуемые JAR-файлы можно скачать на соответствующую машину и включить их, использовав параметр `--jars` или указав координаты Maven в параметре `--packages`. А поскольку командная оболочка Spark запускается так же, можно воспользоваться тем же синтаксисом для включения MySQL JDBC JAR в примере 3.35.

Пример 3.35. Включаем MySQL JDBC JAR

```
spark-submit --jars ./resources/mysql-connector-java-5.1.38.jar $ASSEMBLY_JAR $CLASS
```

В первых версиях фреймворка Spark параметр --jar не позволял включать JAR в переменную classpath драйвера. При использовании в кластере подобной версии Spark необходимо также указать тот же JAR в параметре --driver-class-path.

[1] Отдельные типы могут оказаться некорректно реализованными для некоторых баз данных.

Класс `JdbcDialects` (https://github.com/apache/spark/blob/master/sql/core/src/main/scala/org/apache/spark/sql/jdbc/JdbcDialects.scala) позволяет Spark правильно устанавливать соответствие типов JDBC типам Spark SQL. Если для вашей СУБД отсутствует `JdbcDialect`, то будет применяться диалект по умолчанию, который, скорее всего, станет работать в большинстве случаев. Диалект будет выбран автоматически на основе используемого URL JDBC.

Если вы столкнетесь с необходимостью адаптировать JdbcDialect под свою СУБД, то можете или поискать соответствующий пакет среди spark-packages, или расширить класс JdbcDialect и зарегистрировать собственный диалект.

Как и для других встроенных источников данных, существует удобный адаптер для определения свойств, необходимых для загрузки JDBC-данных, показанный в примере 3.36. Этот удобный адаптер `JDBC` принимает на входе URL, таблицу и объект `java.util.Properties` со свойствами соединения (такими как информация об аутентификации). Объект свойств объединяется со свойствами, задаваемыми в самих классах чтения/записи. Хотя объект свойств — обязательный параметр, можно передать пустой объект свойств, и они будут устанавливаться в интерфейсе чтения/записи.

Пример 3.36. Создание набора DataFrame из источника данных JDBC

```
session.read.jdbc("jdbc:dialect:serverName;user=user;password=pass",
  "table", new Properties)

session.read.format("jdbc")
  .option("url", "jdbc:dialect:serverName")
  .option("dbtable", "table").load()
```

API для сохранения набора `DataFrame` очень напоминает API, используемый для загрузки. Указывать путь в функции `save()` не требуется, поскольку вся информация уже задана, как показано в примере 3.37.

Пример 3.37. Запись объекта DataFrame в источник данных JDBC

```
df.write.jdbc("jdbc:dialect:serverName;user=user;password=pass",
  "table", new Properties)

df.write.format("jdbc")
  .option("url", "jdbc:dialect:serverName")
  .option("user", "user")
  .option("password", "pass")
  .option("dbtable", "table").save()
```

Помимо чтения из источников данных JDBC и записи в них, у Spark SQL есть собственный JDBC-сервер (описанный в разделе «Сервер JDBC/ODBC» далее в этой главе).

Parquet

Apache Parquet — распространенный формат, непосредственно поддерживаемый Spark SQL, чрезвычайно эффективно использующий место и очень популярный. Своей известностью он обязан множеству возможностей, которые включают удобное разбиение на несколько файлов, сжатие, вложенные типы и многое другое, описанные в тематической документации (https://parquet.apache.org/documentation/latest/). Вследствие популярности Parquet в фреймворк Spark были добавлены дополнительные параметры для чтения и записи файлов в этом формате. Они перечислены в табл. 3.10. В отличие от сторонних источников данных их настройка в основном производится через объект `SQLContext`, хотя некоторые могут быть заданы как в `SQLContext`, так и в интерфейсах `DataFrameReader/DataFrameWriter`.

Таблица 3.10. Параметры источника данных Parquet[1]

SQLConf	Параметр DataFrameReader/ DataFrameWriter	По умолчанию	Назначение
spark.sql.parquet. mergeSchema	mergeSchema	False	Определяет, необходимо ли объединять схемы секций при чтении. Может оказаться весьма затратной, так что отключена по умолчанию, начиная с версии 1.5.0
spark.sql.parquet. binaryAsString	N/A	False	Рассматривает бинарные данные как строки. Старые версии фреймворка Spark записывают строки в виде бинарных данных
spark.sql.parquet. cacheMetadata	N/A	True	Кэширует метаданные Parquet. Обычно эта операция безопасна, если нижележащие данные не модифицируются другим процессом
spark.sql.parquet. compression.codec	N/A	Gzip	Задает используемый для данных Parquet кодек сжатия. Допустимые варианты: uncompressed, snappy, gzip и lzo
spark.sql.parquet. filterPushdown	N/A	True	Выполняет спуск фильтров в Parquet (при возможности)[1]
spark.sql.parquet. writeLegacyFormat	N/A	False	Записывает метаданные Parquet в унаследованном формате

Продолжение ⇩

[1] Спуск (pushdown) означает выполнение вычислений на уровне хранилища, так что в случае Parquet это часто может значить пропуск ненужных строк или файлов.

Таблица 3.10 *(продолжение)*

SQLConf	Параметр DataFrameReader/ DataFrameWriter	По умолчанию	Назначение
spark.sql.parquet. output.committer. class	N/A	org.apache. parquet.hadoop. ParquetOutput- Committer	Класс, используемый Parquet для фиксации выводимых данных. При записи в Amazon S3, возможно, имеет смысл попробовать org.apache.spark.sql.parquet.Di- rectParquetOutputCommitter

Чтение данных в формате Apache Parquet из старых версий фреймворка Spark требует использования некоторых особых параметров, как показано в примере 3.38.

Пример 3.38. Чтение файла в формате Apache Parquet, записанного старой версией Spark

```
def loadParquet(path: String): DataFrame = {
  // Настройка Spark для чтения бинарных данных как строки
  // Примечание: должно быть настроено на уровне сеанса
  session.conf.set("spark.sql.parquet.binaryAsString", "true")

  // Загружаем данные в формате parquet с применением схемы слияния
  // (настраивается с помощью параметра)
  session.read
    .option("mergeSchema", "true")
    .format("parquet")
    .load(path)
}
```

Запись данных в формате Parquet при настройках по умолчанию не представляет сложности, как показано в примере 3.39.

Пример 3.39. Запись файла в формате Parquet с настройками по умолчанию

```
def writeParquet(df: DataFrame, path: String) = {
  df.write.format("parquet").save(path)
}
```

Таблицы Hive

Возможность взаимодействия с таблицами Hive добавляет еще один параметр. Как описано в подразделе «Простые SQL-запросы в старом стиле и взаимодействие с данными из Hive» на с. 68, один из способов получить данные из таблицы Hive — написать SQL-запрос к ней с получением результата в виде набора `DataFrame`. Интерфейсы чтения и записи класса `DataFrame` можно использовать как с таблицами Hive, так и с остальными источниками данных, как показано в примере 3.40.

Пример 3.40. Загрузка таблицы Hive

```
def loadHiveTable(): DataFrame = {
  session.read.table("pandas")
}
```

При загрузке таблицы Hive модуль Spark SQL преобразует метаданные и кэширует результат. Если исходные метаданные изменились, то можно воспользоваться командой sqlContext.refreshTable("tablename") для их обновления или отключить кэширование, установив значение параметра spark.sql.parquet.cacheMetadata равным false.

Сохранение управляемой таблицы, продемонстрированное в примере 3.41, несколько отличается.

Пример 3.41. Запись управляемой таблицы

```
def saveManagedTable(df: DataFrame): Unit = {
  df.write.saveAsTable("pandas")
}
```

За исключением особых случаев, результаты сохраняются в таблице Hive в ориентированном на Spark формате, который могут не понимать другие утилиты.

Наборы RDD

Наборы `DataFrame` Spark SQL можно легко преобразовывать в наборы RDD объектов типа `Row`. Кроме того, они легко создаются из наборов RDD объектов типа `Row`, Java-компонентов, case-классов Scala и кортежей. При работе с наборами RDD строк в формате JSON уместны методы, обсуждавшиеся в пункте «JSON» на с. 72. `Dataset` типа T можно легко преобразовать в RDD типа T, что служит удобным связующим звеном между объектами `Dataset` и RDD конкретных case-классов (вместо объектов `Row`). Наборы RDD — частный случай источников данных, поскольку при перемещении из/в RDD данные остаются в пределах Spark, а не записываются во внешнюю систему и не читаются из нее.

Конвертация типа из набора DataFrame в набор RDD — преобразование (а не действие); однако преобразование типа из набора в DataFrame или Dataset может включать вычисление входного RDD (или выборку некоторой его части).

Создание набора DataFrame из набора RDD в общем случае влечет за собой накладные расходы. Данные нужно преобразовывать во внутренний формат Spark SQL.

При создании `DataFrame` из RDD модулю Spark SQL приходится добавлять информацию о схеме. Создавая `DataFrame` из набора RDD case-классов или простых Java-объектов в старом стиле (POJO), Spark SQL может воспользоваться рефлексией, чтобы автоматически определить схему, как показано в примере 3.42. Можно также вручную задать схему для данных с помощью структуры, обсуждавшейся в разделе «Основные сведения о схемах» на с. 52. Особенно полезно это может оказаться в том случае, когда часть полей данных не допускает неопределенных значений. Если Spark SQL не способен определить схему данных, используя рефлексию, то необходимо задать ее самостоятельно, скажем, в виде набора RDD-объектов типа `Row` (допустим, путем вызова `.rdd` объекта `DataFrame` для функционального преобразования, как показано в примере 3.42).

Пример 3.42. Создание объектов DataFrame из наборов RDD

```
def createFromCaseClassRDD(input: RDD[PandaPlace]) = {
  // Создаем объект DataFrame явным образом, с помощью сеанса
  // и логического вывода схемы
  val df1 = session.createDataFrame(input)

  // Создаем объект DataFrame с помощью неявных функций сеанса
  // и логического вывода схемы
  val df2 = input.toDF()

  // Создаем набор RDD объектов типа Row из нашего RDD case-классов
  val rowRDD = input.map(pm => Row(pm.name,
    pm.pandas.map(pi => Row(pi.id, pi.zip, pi.happy, pi.attributes))))

  val pandasType = ArrayType(StructType(List(
    StructField("id", LongType, true),
    StructField("zip", StringType, true),
    StructField("happy", BooleanType, true),
    StructField("attributes", ArrayType(FloatType), true))))

  // Создаем объект DataFrame с заданной схемой явным образом
  val schema = StructType(List(StructField("name", StringType, true),
    StructField("pandas", pandasType)))

  val df3 = session.createDataFrame(rowRDD, schema)
}
```

Case-классы или Java-компоненты, описанные внутри другого класса, иногда могут вызывать проблемы. При сбое преобразования типа RDD лучше убедиться, что используемый case-класс не объявлен внутри другого класса.

Преобразовать тип из `DataFrame` в RDD чрезвычайно просто, однако при этом получается набор RDD объектов типа `Row`, как показано в примере 3.43. Поскольку строка может содержать произвольные данные, необходимо указать тип (или выполнить приведение итогового типа) при извлечении значений из каждого столбца строки. С помощью объектов `Dataset` можно непосредственно получить обратно набор RDD, шаблонизированный тем же типом данных, что позволит значительно упростить обратное преобразование типа в удобный RDD.

Хотя в языке Scala есть множество функций неявного преобразования различных числовых типов данных, они обычно неприменимы в Spark SQL, так что приходится пользоваться явным приведением типов.

Пример 3.43. Преобразование типа DataFrame

```
def toRDD(input: DataFrame): RDD[RawPanda] = {
  val rdd: RDD[Row] = input.rdd
  rdd.map(row => RawPanda(row.getAs[Long](0), row.getAs[String](1),
    row.getAs[String](2), row.getAs[Boolean](3), row.getAs[Array[Double]](4)))
}
```

Если вы заранее знаете, что схема вашего объекта DataFrame совпадает со схемой другого, то можете воспользоваться существующей схемой при создании нового DataFrame. Подобное действие часто выполняется при преобразовании типа из входного объекта DataFrame в RDD для функциональной фильтрации и при последующем обратном преобразовании.

Локальные коллекции

Подобно наборам RDD, наборы `DataFrame` можно создавать из локальных коллекций и возвращать в виде последних, как показано в примере 3.44. Требования к оперативной памяти не меняются: все содержимое объекта `DataFrame` должно размещаться в оперативной памяти драйверной программы. По сути, использование локальных коллекций ограничено модульными тестами и соединением небольших наборов данных с большими распределенными наборами данных.

Пример 3.44. Создание набора DataFrame из локальной коллекции

```
def createFromLocal(input: Seq[PandaPlace]) = {
  session.createDataFrame(input)
}
```

Мы воспользовались тут API LocalRelation, что дало возможность задать схему таким же образом, как и при преобразовании типа из набора RDD в DataFrame.

В более ранних, чем 1.6, версиях PySpark схема определяется на основе только первой записи.

Отправка данных обратно с помощью метода `collect()` в виде локальной коллекции чаще всего производится после агрегирования или фильтрации данных. Например, как в случае отправки конвейером ML коэффициентов или (как вы увидите в примере со Златовлаской в главе 6) квантилей на драйвер. В примере 3.45 мы продемонстрируем способ локального применения метода `collect()` к набору `DataFrame`. В случае больших наборов данных рекомендуется сохранять данные во внешней системе хранения (такой как база данных или HDFS).

Как и в случае с RDD, не следует отправлять большие наборы DataFrame обратно на драйвер. Пользователям языка Python важно не забывать, что метод toPandas() собирает данные локально.

Пример 3.45. Локальный сбор итоговых данных

```
def collectDF(df: DataFrame) = {
  val result: Array[Row] = df.collect()
  result
}
```

Дополнительные форматы

Что касается Spark Core, поставляемые непосредственно со Spark форматы данных охватывают лишь верхушку айсберга доступных для взаимодействия типов систем. Некоторые поставщики публикуют собственные реализации, и многие из них опубликованы на сайте Spark Packages (https://spark-packages.org/). На момент написания книги на странице источников данных (https://spark-packages.org/?q=tags%3A%22Data%20Sources%22) перечислено более 20 форматов, наиболее популярными из которых являются Avro (https://spark-packages.org/package/databricks/spark-avro), Redshift (https://spark-packages.org/package/databricks/spark-redshift), CSV (https://spark-packages.org/package/databricks/spark-csv)[1] и универсальный адаптер deep-spark (https://spark-packages.org/package/Stratio/deep-spark) для более чем шести СУБД.

Пакеты Spark можно включить в приложение несколькими различными способами. На этапе анализа (скажем, при использовании командной оболочки) можно включить их путем указания параметра `--packages` в командной строке, как в примере 3.46. Этот же подход подойдет при отправке приложений с помощью команды `spark-submit`, но информация касается только пакетов во время выполнения, а не во время компиляции. Для добавления пакетов во время компиляции можно доба-

[1] Сейчас пакет spark-csv уже включен в состав Spark 2.0.

вить в сборку координаты Maven. При компоновке с помощью `sbt` работу с зависимостями пакетов способны упростить плагин sbt-spark-package (https://github.com/databricks/sbt-spark-package) и переменная `spDependencies`. В остальных случаях прекрасно подойдет перечисление их вручную, как в примере 3.47.

Пакет Spark CSV в настоящий момент включен в состав Spark 2.0+, так что включать его необходимо только для более ранних версий.

Пример 3.46. Запуск командной оболочки Spark с поддержкой CSV

```
./bin/spark-shell --packages com.databricks:spark-csv_2.11:1.5.0
```

Пример 3.47. Включение пакета spark-csv в качестве sbt-зависимости

```
"com.databricks" % "spark-csv_2.11" % "1.5.0"
```

После включения стороннего пакета в задание Spark необходимо указать формат, точно так же, как мы поступали с пакетами самого фреймворка. Имя пакета можно найти в его документации. Например, строка формата для `spark-csv` выглядит как `com.databricks.spark.csv`. Для встроенного CSV-формата (в Spark 2.0+) вместо этого можно просто указать `csv` (или полное название `org.apache.spark.sql.csv`).

Если требуемый формат данных не поддерживается непосредственно ни Spark, ни одной из библиотек, то возможностей немного. Большинство форматов поддерживаются в виде входных форматов Hadoop, так что можно попытаться загрузить данные в качестве входного формата Hadoop, после чего выполнить преобразование типа получившегося объекта RDD, как обсуждалось в пункте «Наборы RDD» на с. 77. Этот подход относительно прост, но влечет за собой невозможность операций в хранилище данных[1].

Для обеспечения более глубокой интеграции можно реализовать собственный источник данных с помощью API источников данных (https://databricks.com/blog/2015/01/09/spark-sql-data-sources-api-unified-data-access-for-the-spark-platform.html). В зависимости от того, для каких операций необходима поддержка спуска операторов, в BaseRelation понадобится реализовать дополнительные типажи из пакета `org.apache.spark.sql.sources`. Нюансы реализации нового источника данных Spark SQL выходят за рамки данной книги. Но если этот вопрос вас интересует, то рекомендуем начать с документации языка Scala по пакету `org.apache.spark.sql.sources` (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.sources.package) и класса CsvRelation (https://github.com/databricks/spark-csv/blob/master/src/main/scala/com/databricks/spark/csv/CsvRelation.scala) пакета `spark-csv`.

[1] Например, чтения только нужных секций, для которых фильтр соответствует одной из наших схем секционирования.

Режимы сохранения

В базовом Spark для сохранения наборов RDD всегда требуется, чтобы целевой каталог не существовал; это усложняет добавление данных в уже имеющиеся таблицы. С помощью же Spark SQL можно задать желаемое поведение программы при записи по пути, который уже включает данные. Поведение по умолчанию — `SaveMode.ErrorIfExists`. Аналогично поведению объектов RDD Spark генерирует исключение, если целевые данные/таблица уже существуют. Другие варианты режимов сохранения и их поведение перечислены в табл. 3.11. Пример 3.48 иллюстрирует возможности настройки этих альтернативных режимов.

Таблица 3.11. Режимы сохранения

Режим сохранения	Поведение
ErrorIfExists	Генерирует исключение, если целевые данные/таблица уже существуют. Если же не существуют, то записывает данные
Append	Если целевые данные/таблица уже существуют, дописывает туда данные. Если данных еще нет, то записывает их
Overwrite	Если целевые данные/таблица уже существуют, то удаляет их. Записывает данные
Ignore	Если целевые данные/таблица уже существуют, то пропускает этап записи без какого-либо уведомления. В противном случае записывает данные

Пример 3.48. Установка режима сохранения Append

```
def writeAppend(input: DataFrame): Unit = {
  input.write.mode(SaveMode.Append).save("output/")
}
```

Секции (обнаружение и запись)

Секционирование данных — важная составляющая Spark SQL, поскольку оно необходимо для одной из ключевых оптимизаций: чтения только тех данных, которые нужны (этот вопрос мы обсудим в подразделе «Логические и физические планы выполнения» на с. 90). Если в момент записи данных известно, каким образом конечные пользователи будут обращаться к ним (например, читать их на основе почтового индекса), то не помешает применить эти знания для секционирования вывода. При чтении данных полезно знать, как функционирует обнаружение секций, чтобы лучше понимать, получится ли спустить туда фильтр.

Спуск фильтров будет весьма кстати при работе с большими наборами данных, обеспечивая фреймворку Spark возможность обращаться только к необходимому для вычислений подмножеству данных вместо выполнения полного просмотра таблицы.

При чтении секционированных данных достаточно указать Spark корневой каталог данных, и фреймворк автоматически обнаружит различные секции. В качестве

ключей секций можно использовать не все типы данных; в настоящий момент поддерживаются только строки и числовые данные.

Если все данные хранятся в одном объекте `DataFrame`, то можно легко задать информацию о секционировании во время записи данных с помощью API класса `DataFrameWriter`. Функция `partitionBy` принимает в качестве параметра список столбцов, по которым необходимо секционировать результаты, как показано в примере 3.49. Кроме того, можно вручную сохранить отдельные наборы `DataFrame` (например, в момент записи данных из различных заданий), используя отдельные вызовы метода `save`.

Пример 3.49. Сохранение данных, секционированных по почтовому индексу

```
def writeOutByZip(input: DataFrame): Unit = {
  input.write.partitionBy("zipcode").format("json").save("output/")
}
```

Помимо разбиения данных по ключу секционирования, будет полезно убедиться, что размеры итоговых файлов приемлемы, особенно если данные станут использоваться другим заданием Spark далее по ходу потока данных.

Наборы Dataset

Интерфейс `Dataset` — замечательное расширение Spark SQL, обеспечивающее дополнительную проверку типов во время компиляции. Появившись в Spark 2.0, класс `DataFrame` является специализированной версией класса `Dataset`, работающей с универсальными объектами типа `Row`, в которой, следовательно, отсутствует обычная проверка типов во время компиляции, имеющаяся в `Dataset`. API `Dataset` — сильно типизированная коллекция с набором реляционных (`DataFrame`) и функциональных (RDD) преобразований. Подобно объектам `DataFrame`, объекты `Dataset` представлены логическим планом, подходящим для работы оптимизатора Catalyst (см. раздел «Оптимизатор запросов» на с. 90), а при кэшировании данные хранятся во внутреннем формате Spark SQL.

API Dataset появился в Spark 1.6 и будет меняться в последующих версиях. Пользователям этого API рекомендуется рассматривать его как «предварительную версию». Актуальную информацию по API Dataset можно найти в Scaladoc (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset).

Взаимозаменяемость наборов RDD, DataFrame и локальных коллекций

Можно легко преобразовывать объекты класса `Dataset` в объект `DataFrame` или RDD и из них, но в исходном варианте они тоже не допускают непосредственного расширения. Преобразование типа из набора/в набор RDD включает кодирование/декодирование данных в другую форму. Преобразование типа из/в `DataFrame`

обходится почти «бесплатно» в том смысле, что лежащие в его основе данные не нужно менять, добавляется/удаляется только дополнительная информация о типе во время компиляции.

В Spark 2.0 тип DataFrame был заменен псевдонимом типа Dataset[Row].

Псевдоним типа для класса DataFrame невидим в языке Java для Spark 2.0, так что обновление кода на языке Java требует замены DataFrame на Dataset[Row].

Для преобразования типа из `DataFrame` в `Dataset` к объекту `Dataset` можно применить функцию `as[ElementType]` и получить `Dataset[ElementType]`, как показано в примере 3.50. Тип `ElementType` должен быть case-классом или его аналогом, скажем, кортежем, состоящим из типов, доступных для представления модулем Spark SQL (см. раздел «Основные сведения о схемах» на с. 52). Для создания `Dataset` из локальной коллекции существует функция `createDataSet(...)` в объекте `SQLContext` и неявная функция `toDS()` для последовательностей аналогично `createDataFrame(...)` и `toDF()`. Чтобы преобразовать набор RDD в объект `Dataset`, можно сначала выполнить преобразование типа из RDD в `DataFrame`, а затем — в `Dataset`.

Для загрузки данных из объекта Dataset, в случае если источник данных не предоставляет специального API, можно сначала загрузить данные в объект DataFrame, а затем преобразовать его тип в Dataset. Поскольку преобразование типа объекта в Dataset приводит просто к добавлению информации, то проблема немедленного вычисления перед вами не стоит и в дальнейшем можно будет спустить фильтры и другие подобные операции на уровень хранилища данных.

Пример 3.50. Создание объекта Dataset из объекта DataFrame

```
def fromDF(df: DataFrame): Dataset[RawPanda] = {
  df.as[RawPanda]
}
```

Обратное преобразование типа объекта `Dataset` в RDD или `DataFrame` можно выполнить аналогично преобразованию типа объектов `DataFrame`. И то и другое показано в примере 3.51. Функция `toDF()` просто копирует в набор `DataFrame` логический план, используемый в наборе `Dataset`, поэтому не требуется никакой логический вывод схемы или преобразование типа, как при преобразовании из на-

бора RDD. Преобразовать набор `Dataset` объектов типа `T` в набор RDD объектов типа `T` можно, вызвав метод `.rdd`, что, в отличие от вызова `toDF`, включает преобразование данных из внутреннего SQL-формата в обычные типы.

Пример 3.51. Преобразование набора Dataset в набор DataFrame и RDD

```
/**
 * Демонстрация преобразования набора Dataset в набор RDD
 */
def toRDD(ds: Dataset[RawPanda]): RDD[RawPanda] = {
  ds.rdd
}

/**
 * Демонстрация преобразования набора Dataset в набор DataFrame
 */
def toDF(ds: Dataset[RawPanda]): DataFrame = {
  ds.toDF()
}
```

Сильная типизация на этапе компиляции

Одно из преимуществ наборов `Dataset` над обычными наборами `DataFrame` — их сильная типизация на этапе компиляции. У `DataFrame` имеется информация о схеме на этапе выполнения, но нет сведений о ней на этапе компиляции. Такая сильная типизация особенно удобна при создании библиотек, поскольку позволяет задать более четкие требования к входным и возвращаемым типам.

Упрощенные функциональные (RDD-подобные) преобразования

Одно из ключевых преимуществ API Dataset — более простая интеграция с пользовательским кодом на языках Scala и Java. Класс `Dataset` предоставляет методы `filter`, `map`, `mapPartitions` и `flatMap` с такими же сигнатурами функций, как и в классах RDD, но с существенным требованием: возвращаемый тип `ElementType` тоже должен быть понятен Spark SQL (как кортежи или case-классы типов, обсуждавшихся в разделе «Основные сведения о схемах» на с. 52). Пример 3.52 иллюстрирует это с помощью простой функции `map`.

Пример 3.52. Функциональный запрос к набору Dataset

```
def funMap(ds: Dataset[RawPanda]): Dataset[Double] = {
  ds.map{rp => rp.attributes.filter(_ > 0).sum}
}
```

Помимо функциональных преобразований, таких как `map` и `filter`, можно комбинировать реляционные операции и операции группировки/агрегирования.

Реляционные преобразования

Класс `Dataset` предоставляет типизированную версию `select` для реляционных преобразований. При указании параметров этого метода необходимо включать в соответствующие выражения информацию о типах, как показано в примере 3.53. Добавить ее можно, вызвав `as[ReturnType]` для выражения/столбца.

Пример 3.53. Простой реляционный запрос к набору Dataset

```
def squishyPandas(ds: Dataset[RawPanda]): Dataset[(Long, Boolean)] = {
  ds.select($"id".as[Long], ($"attributes"(0) > 0.5).as[Boolean])
}
```

У некоторых операций, например select, имеется как типизированная, так и не типизированная реализация. Если передать вместо TypedColumn просто Column, то будет возвращен объект DataFrame вместо Dataset.

Реляционные преобразования с несколькими Dataset

Помимо преобразований над отдельными `Dataset` существуют также преобразования для работы с несколькими такими объектами. Доступны для использования все стандартные операции над множествами, а именно `intersect`, `union` и `subtract`, с учетом тех же стандартных ограничений, перечисленных в табл. 3.9. Кроме того, поддерживается соединение наборов `Dataset`, но для упрощения работы с информацией о типе возвращаемая структура несколько отличается от случая обычных SQL-соединений.

Группирующие операции с наборами Dataset

Подобно группирующим операциям с наборами `DataFrame` (описанными в пункте «Агрегирующие функции и функция groupBy» на с. 62), операция `groupBy` в момент применения к наборам `Dataset` до Spark 2.0 возвращала (при группировке на основе произвольной функции) объект `GroupedDataset` (http://spark.apache.org/docs/1.6.2/api/scala/index.html#org.apache.spark.sql.GroupedDataset) или `KeyValueGroupedDataset` (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.KeyValueGroupedDataset). При группировке на основе реляционного/`Dataset` DSL-выражения она возвращала `RelationalGroupedDataset` (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.RelationalGroupedDataset). Для них всех можно задать агрегирующие функции, а также воспользоваться функциональным API `mapGroups`. Как и в случае выражения из подраздела «Реляционные преобразования» выше, необходимо применять типизированные выражения, чтобы результат был и объектом `Dataset`.

Мы можем переписать вышеприведенный пример 3.23 по подсчету максимального размера панд в зависимости от почтового индекса так, как показано в примере 3.54.

Существующие для объектов GroupedData удобные функции (например, min, max и т. п.) отсутствуют, так что задавать агрегирующие функции необходимо через agg.

Пример 3.54. Типизированный подсчет максимального размера панд в зависимости от почтового индекса

```
def maxPandaSizePerZip(ds: Dataset[RawPanda]): Dataset[(String, Double)] = {
  ds.map(rp => MiniPandaInfo(rp.zip, rp.attributes(2)))
    .groupByKey(mp => mp.zip).agg(max("size").as[Double])
}
```

Помимо использования типизированных SQL-выражений к агрегированным столбцам с помощью `mapGroups`, можно применять к сгруппированным данным произвольные функции языка Scala, как показано в примере 3.55. Это позволит избежать написания пользовательских агрегирующих функций (UDAF) (обсуждаемых в следующем разделе «Расширение классов пользовательскими функциями, а также пользовательскими функциями агрегирования (UDF, UDAF)»). Хотя писать пользовательские UDAF может быть непросто, они способны помочь добиться лучшей производительности, чем функция `mapGroups`, и их можно использовать и для наборов `DataFrame`.

Пример 3.55. Подсчет максимального размера панд в зависимости от почтового индекса с помощью функции mapGroups

```
def maxPandaSizePerZipScala(ds: Dataset[RawPanda]): Dataset[(String, Double)] = {
  ds.groupByKey(rp => rp.zip).mapGroups{ case (g, iter) =>
    (g, iter.map(_.attributes(2)).reduceLeft(Math.max(_, _)))
  }
}
```

Расширение классов пользовательскими функциями, а также пользовательскими функциями агрегирования (UDF, UDAF)

Пользовательские функции, а также пользовательские функции агрегирования позволяют расширять API класса `DataFrame` и API SQL собственным кодом с сохранением возможности применения оптимизатора Catalyst. API класса `Dataset` (см. раздел «Наборы Dataset» на с. 83) — еще один эффективный вариант использования различных UDF и UDAF. Это хорошо сказывается на производительности, ведь в противном случае приходилось бы преобразовывать данные в наборы RDD (и, вероятно, обратно) для выполнения произвольных функций, что требует немалых затрат ресурсов. К различным UDF и UDAF можно обращаться из обычных SQL-выражений, это позволяет взаимодействовать с ними аналитикам и вообще тем людям, которым привычнее работать с SQL.

Важно отметить, что при использовании UDF и UDAF, написанных на не-JVM-языках, например Python, теряются практически все связанные с производительностью выгоды, ведь данные все равно приходится перемещать из JVM.

Если большую часть кода вы пишете на Python, но хотели бы вызывать UDF без снижения производительности, то можете написать UDF на языке Scala и зарегистрировать их, чтобы использовать в языке Python (как в пакете Sparkling Pandas) (http://sparklingpandas.com/)[1].

Писать неагрегирующие UDF для пакета Spark SQL очень просто: достаточно написать обычную функцию и зарегистрировать ее с помощью метода `sqlContext.udf().register`. В примере 3.56 показана простая UDF для вычисления длины строки. При регистрации UDF, написанных на языках Python или Java, необходимо также указывать возвращаемый тип.

Пример 3.56. UDF для вычисления длины строки

```
def setupUDFs(sqlCtx: SQLContext) = {
  sqlCtx.udf.register("strLen", (s: String) => s.length())
}
```

Даже написанные на JVM-языках UDF работают обычно медленнее, чем работали бы эквивалентные SQL-выражения. В SPARK-14083 (https://issues.apache.org/jira/browse/SPARK-14083) проделана некоторая предварительная работа по синтаксическому разбору байт-кода и генерации SQL-выражений.

Создавать агрегирующие функции (UDAF) несколько сложнее. Вместо написания обычной функции языка Scala приходится расширять класс `UserDefinedAggregateFunction` и реализовывать несколько различных функций наподобие введения функций агрегирования по ключу для наборов RDD, но работающих с различными структурами данных. Будучи довольно сложными в написании, пользовательские функции агрегирования демонстрируют хорошую производительность по сравнению с такими вариантами, как функции `mapGroups` для класса `Dataset` или даже просто `aggregateByKey` для наборов RDD. Можно или вызвать UDAF непосредственно для столбцов, или добавить ее в реестр функций, как мы делали для неагрегирующих UDF.

[1] На момент написания данной книги разработка проекта Sparkling Pandas заморожена, но в предварительных выпусках есть интересные примеры использования JVM-кода из языка Python.

Пример 3.57 представляет собой простую UDAF для вычисления среднего значения, хотя на практике вы, вероятно, предпочтете использовать встроенную функцию `avg` фреймворка Spark.

Пример 3.57. UDAF для вычисления среднего значения

```
def setupUDAFs(sqlCtx: SQLContext) = {
  class Avg extends UserDefinedAggregateFunction {
    // Входной тип
    def inputSchema: org.apache.spark.sql.types.StructType =
      StructType(StructField("value", DoubleType) :: Nil)
    def bufferSchema: StructType = StructType(
      StructField("count", LongType) ::
      StructField("sum", DoubleType) :: Nil
    )

    // Возвращаемый тип
    def dataType: DataType = DoubleType

    def deterministic: Boolean = true

    def initialize(buffer: MutableAggregationBuffer): Unit = {
      buffer(0) = 0L
      buffer(1) = 0.0
    }

    def update(buffer: MutableAggregationBuffer,input: Row): Unit = {
      buffer(0) = buffer.getAs[Long](0) + 1
      buffer(1) = buffer.getAs[Double](1) + input.getAs[Double](0)
    }

    def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
      buffer1(0) = buffer1.getAs[Long](0) + buffer2.getAs[Long](0)
      buffer1(1) = buffer1.getAs[Double](1) + buffer2.getAs[Double](1)
    }

    def evaluate(buffer: Row): Any = {
      buffer.getDouble(1) / buffer.getLong(0)
    }
  }
  // Необязательная регистрация
  val avg = new Avg
  sqlCtx.udf.register("ourAvg", avg)
}
```

Это несколько сложнее обычной UDF, поэтому обсудим, что делают отдельные части данного кода. Мы начинаем с описания типа входных данных, после чего задаем схему буфера, который собираемся использовать для обрабатываемых в текущий момент данных. Указанные схемы задаются так же, как и схемы для наборов `DataFrame` и `Dataset`, обсуждавшиеся в разделе «Основные сведения о схемах» текущей главы.

Оставшиеся функции — реализации тех же функций, которые используются при написании преобразования `aggregateByKey` над наборами RDD, но вместо получения на входе произвольных объектов Scala мы имеем дело с объектами классов `Row` и `MutableAggregationBuffer`. Итоговая функция `evaluate` принимает на входе объект `Row`, представляющий собой данные для агрегирования, и возвращает конечный результат.

UDF, UDAF и объекты `Dataset` обеспечивают способы совместного применения произвольного кода вместе с Spark SQL.

Оптимизатор запросов

Catalyst — оптимизатор запросов модуля Spark SQL, используемый для преобразования плана запроса в план выполнения, подходящий для работы Spark. Подобно тому как выполнение преобразований над наборами RDD приводит к построению DAG, применение реляционных и функциональных преобразований к наборам `DataFrame`/`Dataset` приводит к тому, что Spark SQL строит дерево, соответствующее плану запроса, называемое логическим планом. Spark может выполнять над этим логическим планом различные оптимизации, а также выбирать между несколькими физическими планами для одного логического на основе стоимостной модели.

Логические и физические планы выполнения

Логический план, формируемый при совершении преобразований над объектами `Dataset`/`DataFrame` (или при SQL-запросах), сначала является неразрешенным. Аналогично компилятору оптимизатор Spark работает поэтапно и до выполнения любых оптимизаций должен разрешить ссылки и типы выражений.

Этот уже разрешенный план называют логическим, и именно в него Spark вносит все упрощения, получая в результате оптимизированный логический план.

Указанные упрощения, например как при сложении двух литералов, можно записать с помощью функций поиска по шаблону на деревьях. Оптимизатор не ограничивается подобным поиском, и правила могут включать произвольный код на языке Scala.

После оптимизации логического плана фреймворк Spark может сформировать физический. Этап данного плана способен включать оптимизации на основе как правил, так и стоимости ради получения в результате оптимального физического плана. Одна из важнейших оптимизаций на данном этапе — спуск предикатов на уровень источника данных.

Генерация кода

В качестве финального этапа Spark может выполнить генерацию кода компонентов. Данная операция производится с помощью Janino, компилирующего код на языке Java. Предыдущие версии использовали Quasi Quotes[1], однако накладные расходы оказались слишком велики, чтобы сделать возможной генерацию кода для малых наборов данных. В некоторых TPCDS-запросах генерация кода способна приводить более чем к десятикратному повышению производительности.

В некоторых ранних версиях фреймворка Spark генерация кода для сложных запросов могла приводить к сбоям. Если вы работаете со старой версией Spark и столкнулись с неожиданным сбоем, то имеет смысл отключить генерацию кода, установив значение false для параметра spark.sql.codegen или spark.sql.tungsten.enabled (в зависимости от версии).

Большие планы запросов и итеративные алгоритмы

Хотя возможности оптимизатора Catalyst весьма широки, в случае очень больших планов запросов в настоящий момент у него возникают проблемы. Эти планы обычно являются результатом использования итеративных алгоритмов, таких как алгоритмы на графах или алгоритмы машинного обучения. Один из простых обходных путей решения указанной проблемы — преобразование типа данных в RDD, а затем обратно в `DataFrame`/`Dataset` в конце каждой итерации, как показано в примере 3.58. Но если вы работаете на языке Python, то следует использовать базовый набор RDD Java вместо двойного преобразования типов с помощью Python (как это сделать, вы можете узнать из примера 7.5). Хотя есть и несколько более громоздкое решение: записать данные в хранилище и продолжать работу там.

Пример 3.58. Преобразование типа в RDD и обратно для сокращения плана запроса

```
val rdd = df.rdd
rdd.cache()
sqlCtx.createDataFrame(rdd, df.schema)
```

Состояние этой проблемы отслеживается в SPARK-13346 (https://issues.apache.org/jira/browse/SPARK-13346), и вы можете взглянуть на обходной путь, используемый в GraphFrames (https://github.com/graphframes/graphframes/blob/c55ad42db45ab31623167658bf32637bb7340838/src/main/scala/org/graphframes/lib/AggregateMessages.scala).

[1] Scala Quasi Quotes — часть макросистемы языка Scala.

Отладка запросов Spark SQL

Хотя у оптимизатора запросов Spark SQL есть доступ к гораздо большему объему информации, чем у нас, иногда бывает нужно «заглянуть за кулисы» и убедиться, что все работает в запланированном режиме. Аналогично `toDebugString` для наборов RDD, для объектов `DataFrame` доступны функции `explain` и `printSchema`.

Может оказаться очень важно знать, получилось ли у Spark SQL спустить фильтр. В ранних версиях Spark SQL этот процесс не всегда проходил так, как ожидалось, поэтому иногда приходилось переставлять фильтр, чтобы он оказался прямо рядом с загрузкой данных. В более новых версиях причиной сбоя спуска фильтра скорее будет неправильная конфигурация источника данных.

Сервер JDBC/ODBC

Модуль Spark SQL предоставляет JDBC-сервер с целью позволить внешним утилитам (например, GUI для бизнес-аналитики) работать с доступными в Spark данными и использовать ресурсы совместно. JDBC-сервер модуля Spark SQL требует, чтобы фреймворк Spark был собран с поддержкой Hive.

В силу того что сервер «живет» обычно довольно долго и работает с одним контекстом, он может помочь удобно организовывать совместное использование кэшируемых таблиц несколькими пользователями.

JDBC-сервер модуля Spark SQL основан на HiveServer2 СУБД Hive, и наиболее подходящие из разработанных для HiveServer2 коннекторов можно использовать непосредственно с JDBC-сервером Spark SQL. Специальные драйверы для Spark SQL предлагает также компания Simba (https://www.simba.com/drivers/spark-jdbc-odbc/#documentation_content).

Запустить этот сервер можно из командной строки или с помощью существующего объекта `HiveContext`. Команды запуска и остановки для командной строки: `./sbin/start-thriftserver.sh` и `./sbin/stop-thriftserver.sh`. При запуске из командной строки можно настраивать различные свойства Spark SQL, задав параметр `--hiveconf свойство=значение` в этой строке. Большая часть остальных параметров командной строки аналогичны параметрам команды `spark-submit`. Хост и порт по умолчанию — `localhost:10000`, их можно настраивать, используя свойства `hive.server2.thrift.port` и `hive.server2.thrift.bind.host`.

При запуске JDBC-сервера с помощью уже существующего HiveContext можно просто обновить свойства конфигурации контекста вместо определения параметров командной строки.

Примеры 3.59 и 3.60 демонстрируют два различных способа конфигурации порта, используемого сервером Thrift.

Пример 3.59. Запуск JDBC-сервера на другом порте

```
./sbin/start-thriftserver.sh --hiveconf hive.server2.thrift.port=9090
```

Пример 3.60. Запуск JDBC-сервера на другом порте в коде на языке Scala

```
hiveContext.setConf("hive.server2.thrift.port", "9090")
HiveThriftServer2.startWithContext(hiveContext)
```

При запуске JDBC-сервера с уже существующим HiveContext не забудьте завершить его работу при выходе.

Резюме

Соображения относительно того, использовать наборы RDD или `DataFrame`/`Dataset`, быстро меняются по мере развития модуля Spark SQL. Задействовать данный модуль может быть непросто, в частности, в случае изменения количества необходимых для различных частей конвейера секций или при желании как-то еще управлять секционированием. Хотя наборам RDD недостает оптимизатора Catalyst и реляционных запросов, они способны работать с множеством типов данных и обеспечивают более непосредственный контроль над некоторыми типами операций. Наборы `DataFrame` и `Dataset` также работают только с ограниченным подмножеством типов данных, — но если ваши данные относятся к одному из этих поддерживаемых классов, то привносимое оптимизатором Catalyst повышение производительности более чем оправдывает упомянутые ограничения.

Наборы `DataFrame` уместны при наличии в основном реляционных преобразований, которые при необходимости можно расширить пользовательскими функциями (UDF). По сравнению с наборами RDD, у `DataFrame` есть преимущество применения эффективного формата хранения Spark SQL, оптимизатора Catalyst и возможности выполнять некоторые операции непосредственно с сериализованными данными. Один из недостатков работы с наборами `DataFrame` — отсутствие сильной типизации во время компиляции, что способно привести к ошибкам неправильного доступа к столбцам и прочим элементарным ошибкам.

Наборы `Dataset` можно использовать при необходимости комбинировать функциональные и реляционные преобразования с сохранением выгод оптимизаций наборов `DataFrame`. Соответственно, во многих случаях это отличная альтернатива наборам RDD. Как и в случае с наборами RDD, наборы `Dataset` параметризуются типом содержащихся в них данных. Благодаря этому возможна сильная проверка типов во время компиляции, однако необходимо знать тип данных во время

компиляции (хотя можно задействовать `Row` или другие универсальные типы). Обеспечение наборами `Dataset` дополнительной безопасности типов способно принести выгоду даже тем приложениям, которым не требуется именно функциональность наборов `DataFrame`. Один из потенциальных недостатков — API Dataset продолжает развиваться, вследствие чего может понадобиться вносить изменения в код по мере обновления до новых версий фреймворка Spark.

Чистые наборы RDD отлично работают с данными, которые не подходят для оптимизатора Catalyst. У этих наборов есть обширный и стабильный функциональный API, а обновления до новых версий фреймворка Spark вряд ли потребуют существенных изменений кода. Наборы RDD также облегчают секционирование, что очень полезно для многих распределенных алгоритмов. Некоторые виды операций (например, многостолбцовые агрегирующие функции, сложные соединения и оконные операции) иногда непросто выразить с помощью API наборов RDD. Наборы могут работать с любыми сериализуемыми данными Java или Kryo, хотя сериализация зачастую требует больших затрат и менее эффективна в смысле расходуемого места на диске по сравнению с ее эквивалентом в наборах `DataFrame/Dataset`.

Теперь, как следует разобравшись с модулем Spark SQL, можно перейти к соединениям как для наборов RDD, так и в Spark SQL.

4 Соединения (SQL и Core)

Соединение данных — важная составляющая многих конвейеров; Spark Core и Spark SQL поддерживают одни и те же базовые типы соединений. Хотя соединения широко распространены и эффективны, при их использовании следует обратить особое внимание на производительность, поскольку они могут потребовать больших объемов перемещения данных по сети или даже создания наборов данных, превышающих наши возможности обработки[1]. В Spark Core особенно важен правильный порядок операций, ведь DAG-оптимизатор, в отличие от SQL-оптимизатора, не способен упорядочить фильтры заново или спустить их на уровень хранилища.

Соединения Spark Core

В этом разделе мы рассмотрим соединения для различных типов наборов RDD. Соединения вообще весьма затратные операции, ведь соответствующие ключи всех RDD должны располагаться в одной и той же секции для возможности соединять их локально. При отсутствии в двух наборах RDD информации о секционировании их придется перетасовать, чтобы метод секционирования (объект `Partitioner`) у них оказался одинаковым и данные с одинаковыми ключами находились в одних секциях, как показано на рис. 4.1. Если объект `Partitioner` у них один, то данные можно разместить совместно, как на рис. 4.2, чтобы избежать необходимости передавать их по сети. Вне зависимости от того, одинаковы ли объекты `Partitioner`, если у одного (или обоих) наборов RDD объект `Partitioner` известен, то создается только узкая зависимость, как на рис. 4.3. Как и при большинстве операций с данными типа «ключ — значение», стоимость соединения растет пропорционально количеству ключей и расстоянию, на которое следует переместить записи, чтобы они попали в нужную секцию.

---

[1] Как говорится, декартово произведение двух больших наборов данных равняется ошибке нехватки памяти.

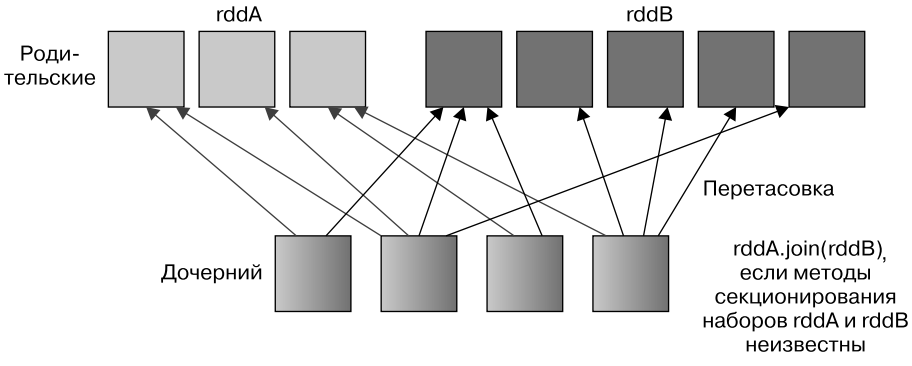

Рис. 4.1. Соединение с перетасовкой

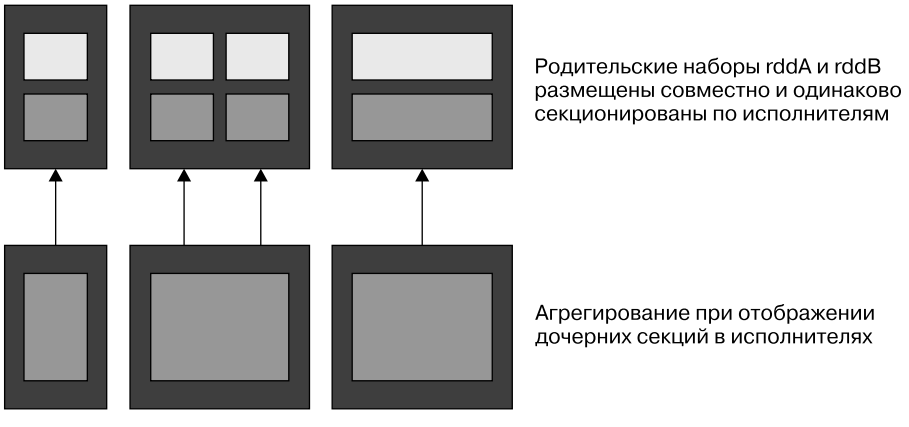

Рис. 4.2. Совместное соединение

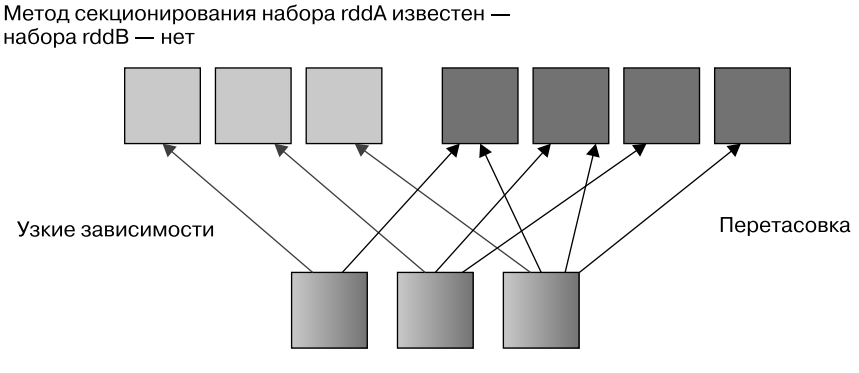

Рис. 4.3. Соединение с двумя[1] известными объектами Partitioner

[1] В данном случае с одним. — *Примеч. пер.*

Два набора RDD будут размещены совместно, если у них одинаковый объект Partitioner и они были подвергнуты перетасовке в рамках одного действия.

Соединения Spark Core реализованы с помощью функции cogroup. Мы обсудим ее в пункте «Совместная группировка» на с. 165.

Выбор типа соединения

В Spark соединение по умолчанию включает только значения для ключей, присутствующих в обоих наборах RDD, а в случае нескольких значений для одного ключа возвращает все возможные перестановки пары «ключ — значение». Лучше всего обычное соединение работает, когда оба набора RDD содержат один и тот же набор неповторяющихся ключей. В случае дублирования ключей объемы данных могут резко возрастать, приводя к проблемам с производительностью, а если один из ключей отсутствует хотя бы в одном из наборов, то соответствующая строка данных будет утрачена. Вот несколько рекомендаций.

- Если в обоих наборах RDD есть повторяющиеся ключи, то соединение может привести к резкому росту объема данных. В этом случае лучше будет выполнить операцию `distinct` или `combineByKey`, чтобы снизить занимаемое ключами пространство, либо воспользоваться функцией `cogroup` для обработки повторяющихся ключей вместо получения полного декартова произведения. С помощью интеллектуального секционирования во время этапа объединения можно избежать второй перетасовки при соединении (мы обсудим это подробнее позже).
- При отсутствии ключа в одном из наборов возникает риск неожиданно потерять часть данных. Безопаснее в таком случае было бы использовать внешнее соединение, чтобы гарантированно сохранить все данные как из левого, так и из правого набора RDD, и отфильтровать данные после соединения.
- Если в одном из наборов RDD имеется удобное для описания подмножество ключей, то, возможно, будет лучше отфильтровать другой набор или выполнить его свертку, чтобы предотвратить масштабную перетасовку данных, которые все равно придется отбросить.

Соединение — одна из самых ресурсоемких операций среди часто используемых в Spark, поэтому лучше перед соединением сделать все возможное для сокращения объема данных.

Допустим, у нас есть один набор RDD с данными вида `(Panda id, score)` и другой — вида `(Panda id, address)` и мы хотели бы отправить каждой панде сообщение

по электронной почте, содержащее ее максимальную оценку. Для этого можно соединить наборы RDD по `id`, после чего вычислить максимальную оценку для каждого из `address`, как показано в примере 4.1.

Пример 4.1. Простое соединение наборов RDD

```
def joinScoresWithAddress1( scoreRDD : RDD[(Long, Double)],
  addressRDD : RDD[(Long, String )]) : RDD[(Long, (Double, String))]= {
    val joinedRDD = scoreRDD.join(addressRDD)
    joinedRDD.reduceByKey( (x, y) => if(x._1 > y._1) x else y )
}
```

Однако этот подход будет работать, вероятно, медленнее, чем если бы мы сначала выполнили свертку данных, чтобы первый набор данных содержал только по одной строке для каждой панды с ее максимальной оценкой, после чего соединили эти данные с данными об адресах (как показано в примере 4.2).

Пример 4.2. Предварительная фильтрация перед соединением

```
def joinScoresWithAddress2(scoreRDD : RDD[(Long, Double)],
  addressRDD: RDD[(Long, String)]) : RDD[(Long, (Double, String))]= {
    val bestScoreData = scoreRDD.reduceByKey((x, y) => if(x > y) x else y)
    bestScoreData.join(addressRDD)
}
```

Если количество оценок для каждой панды = 1000, то объем перетасовываемых данных при первом подходе в 1000 раз превышает объем при этом подходе!

При желании можно было также выполнить левое внешнее соединение, чтобы сохранить для обработки все ключи, даже те, которые отсутствуют в правом наборе RDD. Для этого следовало воспользоваться методом `leftOuterJoin` вместо `join`, как в примере 4.3. В фреймворке Spark можно применять и методы `fullOuterJoin` и `rightOuterJoin` в зависимости от того, какие записи хотелось бы оставить. Все отсутствующие значения — `None`, а присутствующие — `Some('x')`.

Пример 4.3. Простое левое внешнее соединение наборов RDD

```
def outerJoinScoresWithAddress(scoreRDD : RDD[(Long, Double)],
  addressRDD: RDD[(Long, String)]) : RDD[(Long, (Double, Option[String]))]= {
    val joinedRDD = scoreRDD.leftOuterJoin(addressRDD)
    joinedRDD.reduceByKey( (x, y) => if(x._1 > y._1) x else y )
  }
```

Выбор плана выполнения

Для соединения данных фреймворку Spark нужно, чтобы соединяемые данные (то есть относящиеся к одному ключу) находились в одной секции. В Spark соединение реализовано по умолчанию как *перетасовочное хеш-соединение* (shuffled hash join). Оно гарантирует: у данных из каждой секции будут те же ключи сек-

ционирования, что и у второго набора данных, с тем же методом секционирования по умолчанию, что и у первого, поэтому ключи с одним значением хеша из обоих наборов данных будут находиться в одной секции. Хотя описанный подход работает без сбоев, он иногда влечет больше затрат ресурсов, чем необходимо, поскольку требует перетасовки. Избежать последней можно при следующих условиях.

1. Объект `Partitioner` обоих наборов RDD известен.
2. Один из наборов данных достаточно мал, чтобы уместиться в оперативной памяти. В таком случае можно выполнить хеш-соединение с трансляцией (что это такое, мы поясним позже).

Обратите внимание: совместное размещение наборов RDD позволяет избежать перемещения данных по сети, а также перетасовки.

Ускорение соединений путем присвоения известного объекта Partitioner

Если нужно совершить перед соединением операцию, требующую перетасовки (такую как `aggregateByKey` или `reduceByKey`), то можно избежать перетасовки. Для этого следует добавить объект типа `HashPartitioner` с тем же количеством секций в виде явного аргумента первой операции и сохранить набор RDD до выполнения соединения. Можно еще ускорить работу примера из предыдущего раздела, воспользовавшись объектом `Partitioner` для данных об адресах в качестве аргумента для метода `reduceByKey`, как в примере 4.4 и на рис. 4.4.

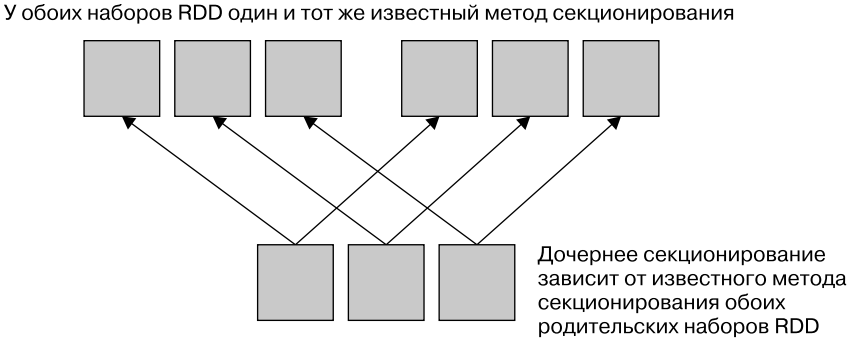

Рис. 4.4. Соединение с двумя известными объектами Partitioner

Пример 4.4. Соединение с известным объектом Partitioner

```
def joinScoresWithAddress3(scoreRDD: RDD[(Long, Double)],
  addressRDD: RDD[(Long, String)]) : RDD[(Long, (Double, String))]= {
  // Если объект Partitioner набора addressRDD известен, воспользуемся им,
  // в противном случае у него имеется объект HashPartitioner
  // по умолчанию, который можно воссоздать, получив количество секций
```

```
  val addressDataPartitioner = addressRDD.partitioner match {
    case (Some(p)) => p
    case (None) => new HashPartitioner(addressRDD.partitions.length)
  }
  val bestScoreData = scoreRDD.reduceByKey(addressDataPartitioner,
     (x, y) => if(x > y) x else y)
  bestScoreData.join(addressRDD)
}
```

Всегда выполняйте сохранение после ресекционирования.

Ускорение соединений с помощью хеш-соединения с трансляцией

Хеш-соединение с трансляцией (broadcast hash join) «выталкивает» один из наборов RDD (меньший) в каждый из рабочих узлов. Там при отображении совершается агрегирование с каждой из секций большего набора. Если один из RDD помещается в оперативной памяти или можно добиться того, чтобы он поместился там, то всегда имеет смысл выполнить хеш-соединение с трансляцией, поскольку оно не требует перетасовки. Иногда (но не всегда) модуль Spark SQL оказывается достаточно «сообразительным» и сам делает такое соединение; в Spark SQL управлять этим можно с помощью параметров `spark.sql.autoBroadcastJoinThreshold` и `spark.sql.broadcastTimeout`. Вышесказанное проиллюстрировано на рис. 4.5.

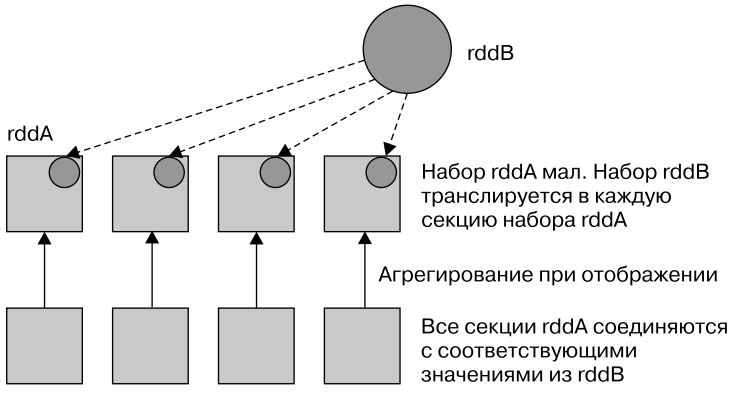

Рис. 4.5. Хеш-соединение с трансляцией

В Spark Core отсутствует реализация хеш-соединения с трансляцией. Вместо этого можно вручную реализовать некую его версию: нужно отправить меньший набор RDD с помощью функции `collect()` на драйвер в виде карты соответствий,

транслировать получившийся результат и применить метод `mapPartitions` для объединения элементов.

Пример 4.5 представляет собой универсальную функцию, предназначенную для использования при соединении большего и меньшего RDD. Ее поведение идентично соединению по умолчанию в Spark. Мы исключаем элементы, ключи которых не присутствуют в обоих наборах.

Пример 4.5. Реализуем вручную хеш-соединение с трансляцией

```
def manualBroadCastHashJoin[K : Ordering : ClassTag, V1 : ClassTag,
V2 : ClassTag](bigRDD : RDD[(K, V1)],
  smallRDD : RDD[(K, V2)])= {
  val smallRDDLocal: Map[K, V2] = smallRDD.collectAsMap()
  bigRDD.sparkContext.broadcast(smallRDDLocal)
  bigRDD.mapPartitions(iter => {
    iter.flatMap{
      case (k,v1 ) =>
        smallRDDLocal.get(k) match {
          case None => Seq.empty[(K, (V1, V2))]
          case Some(v2) => Seq((k, (v1, v2)))
        }
      }
    }, preservesPartitioning = true)
  }
  //end:coreBroadCast[]
}
```

Частично «ручное» хеш-соединение с трансляцией

Иногда не все из меньших наборов RDD умещаются в памяти, а некоторые ключи настолько часто встречаются в большом наборе данных, что оптимально было бы транслировать только самые распространенные ключи. Подобный сценарий особенно полезен, если один из ключей так велик, что не умещается в одной секции. В этом случае для большого набора RDD можно применить метод `countByKeyApprox`[1] с целью понять, трансляция каких ключей принесет больше всего пользы. Затем отфильтровать меньший набор, оставив только эти ключи, и выполнить локальный сбор полученного результата в HashMap. С помощью метода `sc.broadcast` можно транслировать HashMap так, чтобы в каждом из рабочих узлов была только одна копия, после чего вручную выполнить соединение с HashMap. Используя тот же HashMap, можно затем отфильтровать большой набор RDD, исключив наличие в нем большого количества дублирующихся ключей, и выполнить обычное соединение, объединив его с результатом «ручного» соединения. Этот подход весьма запутан, но способен помочь при обработке искаженных данных, которые в противном случае не удалось бы обработать.

[1] Если число различающихся ключей слишком велико, то можно также воспользоваться методом reduceByKey, отсортировать по значениям и применять только k первых.

Соединения Spark SQL

Модуль Spark SQL поддерживает те же базовые типы соединений, что и Spark Core, но оптимизатор существенно облегчает вашу задачу, хотя и лишает части возможностей, позволяющих контролировать происходящее. Например, Spark SQL иногда спускает операции на уровень хранилища или меняет порядок их выполнения ради повышения производительности соединений. С другой стороны, вы не можете контролировать секционирование наборов `DataFrame` или `Dataset`, поэтому не получится вручную избежать перетасовок, как в случае соединений Spark Core.

Соединения наборов DataFrame

Соединение данных из наборов `DataFrame` — одно из чаще всего выполняемых преобразований над несколькими объектами `DataFrame`. При этом все стандартные типы соединений SQL поддерживаются, тип соединения можно задать через параметр `joinType` в вызове метода `df.join(otherDf, sqlCondition, joinType)` в момент соединения.

Как и при соединениях наборов RDD, соединение с неуникальными ключами приводит к декартову произведению в выводимых результатах. (Так что если в левой таблице имеются значения R1 и R2 с ключом key1, а в правой — R3 и R5 с ключом key1, то вы получите в результате (R1, R3), (R1, R5), (R2, R3), (R2, R5).) При изучении соединений Spark SQL мы будем использовать в качестве примера две таблицы панд, табл. 4.1 и 4.2.

Хотя рефлексивные соединения и поддерживаются, необходимо заранее задать различные псевдонимы для интересующих вас полей, чтобы можно было получить к ним доступ.

Таблица 4.1. Таблица размеров панд (левый набор DataFrame)

Имя	**Размер**
Happy	1.0
Sad	0.9
Happy	1.5
Coffee	3.0

Таблица 4.2. Таблица почтовых индексов панд (правый набор DataFrame)

Имя	**Почтовый индекс**
Happy	94110
Happy	94103
Coffee	10504
Tea	07012

Spark поддерживает следующие типы соединений: inner, left_outer (с псевдонимом outer), left_anti, right_outer, full_outer и left_semi. За исключением последнего, все типы предназначены для двух таблиц, но их поведение различается при обработке строк, для которых нет ключей в обеих таблицах.

Внутреннее соединение (inner) используется по умолчанию и представляет собой именно то, что вы обычно называете соединением таблиц. Для него нужно присутствие ключа в обеих таблицах, иначе результат будет отброшен, как видно из примера 4.6 и табл. 4.3.

Пример 4.6. Простое внутреннее соединение

```
// Неявное внутреннее соединение
df1.join(df2, df1("name") === df2("name"))
// Явное внутреннее соединение
df1.join(df2, df1("name") === df2("name"), "inner")
```

Таблица 4.3. Внутреннее соединение наборов df1, df2 по имени панды

Имя	Размер	Имя	Почтовый индекс
Coffee	3.0	Coffee	10504
Happy	1.5	Happy	94110
Happy	1.5	Happy	94103
Happy	1.0	Happy	94110
Happy	1.0	Happy	94103

В результате левого внешнего соединения получается таблица со всеми ключами из левой таблицы. Во всех строках, для которых нет соответствующего ключа в правой таблице, в полях, подлежащих заполнению значений из нее, будет содержаться `null`. Правое внешнее соединение аналогично с точностью до зеркального отражения. В примере 4.7 приведено левое внешнее соединение, результаты которого показаны в табл. 4.4.

Пример 4.7. Левое внешнее соединение

```
// Явное левое внешнее соединение
df1.join(df2, df1("name") === df2("name"), "left_outer")
```

Таблица 4.4. Левое внешнее соединение наборов df1, df2 по имени панды

Имя	Размер	Имя	Почтовый индекс
Sad	0.9	null	null
Coffee	3.0	Coffee	10504
Happy	1.0	Happy	94110
Happy	1.0	Happy	94103
Happy	1.5	Happy	94110
Happy	1.5	Happy	94103

Правое внешнее соединение показано в примере 4.8, а результат его выполнения — в табл. 4.5.

Пример 4.8. Правое внешнее соединение

```
// Явное правое внешнее соединение
df1.join(df2, df1("name") === df2("name"), "right_outer")
```

Таблица 4.5. Правое внешнее соединение наборов df1, df2 по имени панды

Имя	Размер	Имя	Почтовый индекс
Coffee	3.0	Coffee	10504
Happy	1.0	Happy	94110
Happy	1.0	Happy	94103
Happy	1.5	Happy	94110
Happy	1.5	Happy	94103
null	null	Tea	07012

Чтобы сохранить все записи из обеих таблиц, можно воспользоваться полным внешним соединением, в результате которого получается табл. 4.6.

Таблица 4.6. Полное внешнее соединение наборов df1, df2 по имени панды

Имя	Размер	Имя	Почтовый индекс
Sad	0.9	null	null
Coffee	3.0	Coffee	10504
Happy	1.0	Happy	94110
Happy	1.0	Happy	94103
Happy	1.5	Happy	94110
Happy	1.5	Happy	94103
null	null	Tea	07012

Левые полусоединения (как в примере 4.9 и табл. 4.7), а также левые антисоединения (как в табл. 4.8) — единственные типы соединений, в результате которых выводятся значения только из левой таблицы. Левое полусоединение аналогично фильтрации левой таблицы, при которой оставляются только строки с ключами, присутствующими в правой. Левое антисоединение тоже возвращает только данные из левой таблицы, но лишь записи, отсутствующие в правой.

Пример 4.9. Левое полусоединение

```
// Явное левое полусоединение
df1.join(df2, df1("name") === df2("name"), "left_semi")
```

Таблица 4.7. Левое полусоединение

Имя	Размер
Coffee	3.0
Happy	1.0
Happy	1.5

Таблица 4.8. Левое антисоединение

Имя	Размер
Sad	0.9

Рефлексивные соединения

Рефлексивные соединения над наборами `DataFrame` поддерживаются, но при этом возникает дублирование названий столбцов. Поэтому, чтобы обращаться к результатам, необходимо присвоить наборам `DataFrame` различные псевдонимы, иначе к столбцам нельзя будет обратиться из-за конфликта имен (пример 4.10). После присвоения каждому из наборов `DataFrame` псевдонима можно обратиться к отдельным столбцам каждого из `DataFrame` с помощью синтаксиса `ИмяНабора.ИмяСтолбца`.

Пример 4.10. Рефлексивное соединение

```
val joined = df.as("a").join(df.as("b")).where($"a.name" === $"b.name")
```

Хеш-соединения с трансляцией

Узнать тип выполняемого соединения в Spark SQL позволяет вызов `queryExecution.executedPlan`. Как и в случае Spark Core, если одна из таблиц намного меньше другой, то лучше воспользоваться хеш-соединением с трансляцией. Сообщить Spark SQL о необходимости выполнить трансляцию заданного набора при соединении можно, вызвав для этого набора `DataFrame` метод `broadcast` перед соединением (например, `df1.join(broadcast(df2), "key")`). Фреймворк Spark также автоматически применяет параметр `spark.sql.conf.autoBroadcastJoinThreshold`, чтобы определить, следует ли транслировать таблицу.

Соединения наборов Dataset

Соединить наборы `Dataset` можно с помощью метода `joinWith`, ведущего себя аналогично обычному реляционному соединению, за исключением того, что результат представляет собой кортеж различных типов записей, как показано в примере 4.11. Работать с таким результатом после соединения несколько менее удобно, зато

гораздо проще выполнять рефлексивные соединения, как показано в примере 4.12, поскольку не нужно предварительно задавать псевдонимы столбцов.

Пример 4.11. Соединение двух наборов Dataset

```
val result: Dataset[(RawPanda, CoffeeShop)] = pandas.joinWith(coffeeShops,
  $"zip" === $"zip")
```

Пример 4.12. Рефлексивное соединение набора Dataset

```
val result: Dataset[(RawPanda, RawPanda)] = pandas.joinWith(pandas,
  $"zip" === $"zip")
```

Рефлексивное соединение и lit[1](true) позволяют получить декартово произведение набора Dataset самого на себя, что может оказаться полезно, но заодно и демонстрирует, как соединения (особенно рефлексивные) с легкостью приводят к неприемлемым объемам данных.

Как и в случае наборов `DataFrame`, можно задавать тип желаемого соединения (например, inner, left_outer, right_outer, left_semi), управляя обработкой записей, присутствующих только в одном из наборов `Dataset`. Обратите внимание: пропущенные значения будут представлены значениями `null`.

Резюме

Разобравшись с соединениями, сосредоточимся на преобразованиях и связанных с ними вопросах производительности.

[1] Метод, преобразующий литеральное значение в столбец. — *Примеч. пер.*

5 Эффективные преобразования

Чаще всего программы Spark создаются на основе наборов RDD: они включают чтение данных из устойчивого хранилища в формат RDD, выполнение нескольких вычислений и преобразований данных над RDD, а также запись итогового набора RDD в устойчивое хранилище или отправку с помощью метода `collect()` на драйвер. Следовательно, основная сила фреймворка Spark заключается в его преобразованиях: операциях над наборами RDD, возвращающими другие наборы.

В настоящее время Spark содержит специализированные механизмы для почти полудюжины типов RDD, каждый — с собственными свойствами и подборками различных функций преобразований. В этом разделе мы надеемся продемонстрировать инструменты, которые помогут понять, как вычисляются преобразования (или их последовательности). В частности, какие разновидности наборов RDD возвращают эти преобразования, что эффективнее — сохранение наборов или создание контрольных точек между преобразованиями — и как выполнить конкретную последовательность преобразований наиболее эффективно.

В этом разделе речь идет о преобразованиях, связанных с объектами RDD, используемых в Spark Core (и библиотеке MLlib). Наборы RDD также применяются в потоках данных DStreams при работе с Spark Streaming, но их возможности и производительность отличаются. Аналогично большинство обсуждаемых в данной главе функций не поддерживается в наборах DataFrame. В силу иного оптимизатора Spark SQL далеко не все выводы данной главы можно применить к Spark SQL.

По мере развития фреймворка Spark для наборов Dataset становится доступно все больше преобразований, которые можно задействовать в Spark SQL. Они обсуждаются в разделе «Наборы Dataset» на с. 83.

Узкие и широкие преобразования

В главе 2 мы рассказали о важном различии между типами преобразований: с *широкими* и *узкими зависимостями*. Оно является таковым в силу существенных последствий для вычисления этих преобразований и, следовательно, их производительности. В данном разделе мы дадим более четкое определение того, что такое широкие и узкие преобразования, покажем, как выяснить, является ли преобразование узким или широким, и объясним, почему это различие так влияет на выполнение вычислений и производительность.

Напомним, что Spark осуществляет отложенные вычисления, то есть преобразование не выполняется до момента вызова соответствующего действия. Это, как обсуждалось в подразделе «Отложенные вычисления» на с. 29, серьезно влияет на отказоустойчивость, производительность и возможности отладки. Если предыдущие два предложения не вполне понятны, то рекомендуем еще раз заглянуть в главу 2, где в объеме, достаточном для данной главы, приведены основные сведения, необходимые для понимания механизма выполнения Spark.

Подытожим изложенное в главе 2: широкие преобразования — те, для которых требуется перетасовка, а узкие — те, что в ней не нуждаются. В подразделе «Широкие и узкие зависимости» на с. 36 мы поясняли: в узких преобразованиях дочерние секции (секции полученного набора RDD) зависят от известного подмножества родительских секций. Хотя это определение корректно, оно не столь точно, как формальное определение узких преобразований.

Статья 2012 года, в которой впервые была описана семантика вычислений для фреймворка Spark (http://www-bcf.usc.edu/~minlanyu/teach/csci599-fall12/papers/nsdi_spark.pdf), определяет преобразования с узкими зависимостями как такие, в которых «каждая секция родительского набора RDD используется как минимум одной секцией дочернего набора». Создатели Spark описывают преобразования с широкими зависимостями как такие, в которых «от каждой секции родительского набора может зависеть несколько дочерних». Это определение аналогично тому, что мы приводили в главе 2, когда задавали преобразования как узкие или широкие в соответствии с зависимостями дочернего набора RDD. Однако определение создателей Spark устанавливает узкие и широкие зависимости в терминах зависимостей родительского набора, а не дочернего.

Мы полагаем, что представленное в главе 2 определение легче осмыслить, поскольку программы обычно проектируются в направлении от входных данных (родительского набора RDD) к выходным (дочернему набору). Однако механизм вычислений фреймворка Spark (DAG) строит план выполнения в противоположном направлении: от выходных данных (последнего действия) к входному

набору RDD. Следовательно, определение создателей Spark отражает способ вычислений ффреймворка, а значит, является более точным в двух важных смыслах. Во-первых, определение создателей Spark исключает случай одной родительской секции с несколькими дочерними с узкой зависимостью. Это объясняет, почему `coalesce` представляет собой лишь узкое преобразование, хотя и снижает, а не увеличивает количество секций. Во-вторых, определение создателей Spark объясняет, почему количество задач, используемых для выполнения всего вычисления, соотносится скорее с секциями на *выходе*, а не на *входе* — при вычислении набора RDD необходимые для преобразований задачи вычисляются по дочерним секциям.

Рисунок 5.1 демонстрирует зависимости между родительскими и дочерними секциями для узких и широких преобразований в программе Spark из примера 5.1. Допустим, что `rdd1` представляет собой набор RDD целочисленных значений.

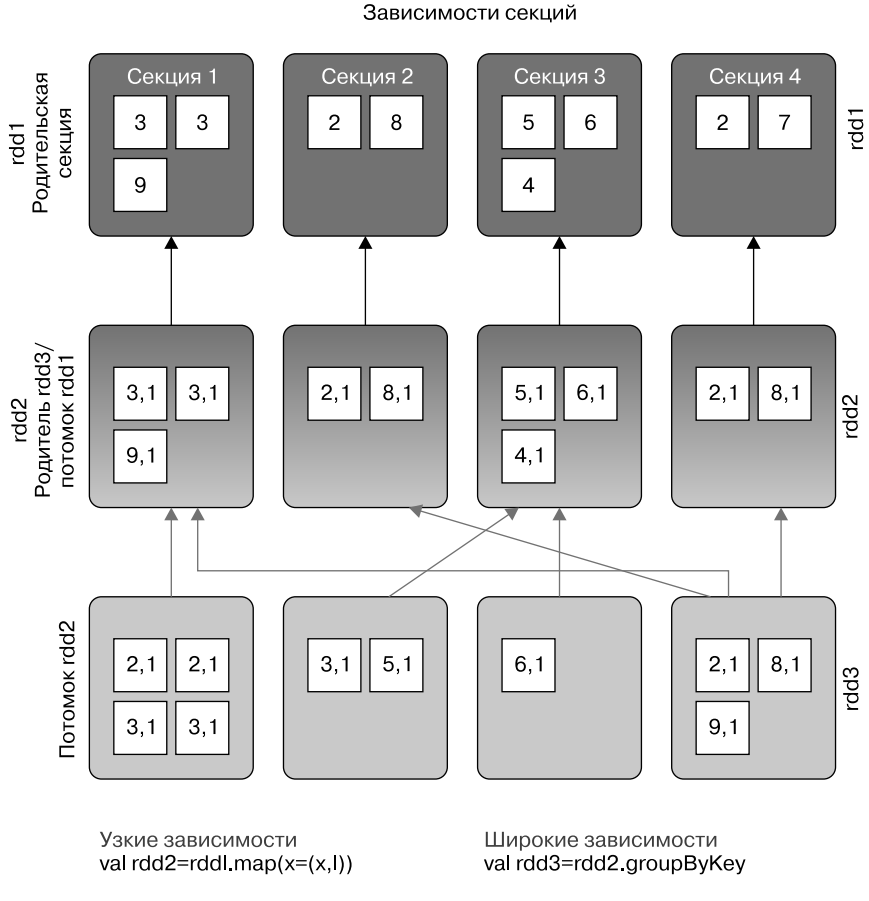

Рис. 5.1. Узкие зависимости между секциями по сравнению с широкими

Пример 5.1. Узкие зависимости по сравнению с широкими

```
// Узкая зависимость. Отображает rdd на кортежи из (x, 1)
val rdd2 = rdd1.map(x => (x, 1))
// Широкая зависимость groupByKey
val rdd3 = rdd2.groupByKey()
```

Структура тут такая же, как и в схемах на рис. 2.2 и 2.3. Стрелки обозначают зависимости между секциями. У всех дочерних секций есть стрелки, указывающие на родительские, от которых они зависят; если стрелка направлена от секции `y` к секции `x`, то `x` зависит от `y`. Черные стрелки обозначают узкие зависимости, а серые — широкие.

Мы предполагаем, что набор RDD состоит из четырех секций. В отличие от схем из главы 2 здесь мы покажем способы распределения по секциям записей в очень маленьком RDD. В данном случае продемонстрируем, как могли бы быть секционированы `RDD1`, `RDD2` и `RDD3`, если бы `RDD1` представлял собой набор RDD из целочисленных значений `3`, `3`, `9`, `2`, `8`, `5`, `6`, `7`.

Как видите, при вычислении шага `map` каждая дочерняя секция зависит ровно от одной родительской, поскольку для выполнения этой операции данные не требуется перемещать между секциями. Однако на шаге `groupByKey` фреймворку Spark приходится переместить некоторое количество записей так, что записи с одинаковым ключом будут находиться в одной секции, с целью получить возможность сгруппировать относящиеся к одному ключу записи в один итератор. (Напоминаем: итератор представляет собой локальную, а не распределенную коллекцию.) Следовательно, дочерние секции зависят от нескольких секций родительского набора RDD.

Эта схема предназначена для демонстрации того, как секции — используемые в вычислительном процессе фреймворка Spark абстрактные концепции — зависят друг от друга, а не от физического перемещения данных между машинами. Строки квадратиков на схеме соответствуют одним и тем же исполнителям в разные моменты времени. Стрелки обозначают зависимости между секциями. На самом деле повторное секционирование данных не обязательно требует перемещения данных между машинами, поскольку секции могут располагаться на одном исполнителе. При смене секции, к которой относится запись, перемещение данных между исполнителями необходимо, ведь записи должны пройти через драйвер, а не передаваться напрямую между исполнителями.

Влияние на производительность

В подразделе «Широкие и узкие зависимости» на с. 36 мы утверждали, что преобразования с узкими зависимостями совершаются быстрее частично вследствие того, что допускают группировку и выполнение за один проход данных. В текущем подразделе мы надеемся пояснить эту мысль с точки зрения выполнения вычислений.

Узкие зависимости не нуждаются в перемещении данных между секциями. Следовательно, эти преобразования не требуют обмена данными с узлом драйвера, и получения одного набора инструкций от драйвера достаточно для совершения произвольного количества таких преобразований над любым подмножеством записей (любой секцией). В терминологии фреймворка Spark любая последовательность узких преобразований может быть вычислена на одном «этапе» плана выполнения запроса.

В отличие от этого, как мы отметили в разделе «Структура задания Spark» на с. 40, связанная с широкой зависимостью перетасовка знаменует собой новый этап вычислений набора RDD. А поскольку задачи должны вычисляться по одной секции, а данные, необходимые для вычисления каждой секции широкой зависимости, могут быть распределены по нескольким машинам, преобразованиям с широкими зависимостями может понадобиться перемещать данные между секциями. А значит, расположенные далее по конвейеру вычисления нельзя произвести до завершения перетасовки.

Например, интуитивно понятно, что выполнить сортировку с помощью узких преобразований невозможно, поскольку она требует упорядочения всех записей, а не только в пределах каждой секции. И действительно, зависимости функции `sortByKey` — широкие. Она требует секционирования данных таким образом, чтобы все ключи в определенном диапазоне располагались в одной секции. Вследствие этого сортировка данных по всем секциям приводит к отсортированному результату. Выполнить узкие преобразования, следующие за сортировкой, невозможно до завершения перетасовки, так как данные в любой из секций могут измениться.

Границы этапов существенно влияют на производительность. За исключением случаев операций с несколькими наборами RDD (например, `join`), относящиеся к обработке одного набора RDD этапы должны выполняться последовательно (см. главу 4). Таким образом, перетасовки ресурсоемки потому, что не только требуют перемещения данных и потенциально операций дискового ввода/вывода (для файлов перетасовок), но и ограничивают параллелизм.

Влияние на отказоустойчивость

Стоимость отказа для секции с широкими зависимостями намного выше, чем для секции с узкими, поскольку приходится заново пересчитывать больше секций. Например, в случае отказа одной секции в родителе набора `mappedRDD` (полученный тип операций `map`) нужно заново пересчитать только одну из ее дочерних секций, причем задачи, необходимые для повторного вычисления данной секции, можно распределить по нескольким исполнителям для ускорения вычислений. Напротив, при потере секции родительского набора отсортированного RDD может понадобиться (в наихудшем сценарии) пересчитать все дочерние секции. Поэтому цена пересчета секции в случае отказа для секций с широкими зависимостями намного выше, чем для секций с узкими.

Организация преобразований с широкими зависимостями цепочкой только повышает риск чрезвычайно дорогостоящих повторных вычислений, особенно если какая-либо из этих зависимостей с высокой вероятностью вызывает ошибки памяти. В некоторых случаях стоимость повторных вычислений может быть настолько высока, что имеет смысл создать контрольные точки (`checkpointing`) для набора RDD ради сохранения промежуточных результатов. Мы обсудим создание этих точек подробнее в разделе «Повторное использование наборов RDD» на с. 135.

Особый случай операции coalesce

Операция `coalesce` используется для изменения количества секций набора RDD. Как показано в схеме на рис. 2.2, при снижении количества выходных секций с помощью операции `coalesce` каждая родительская секция применяется ровно в одной дочерней, поскольку последние представляют собой объединение нескольких родительских. Следовательно, в соответствии с нашим определением узких зависимостей операция `coalesce` является узким преобразованием, несмотря даже на то, что меняет количество секций в наборе RDD. Поскольку задачи решаются в дочерних секциях, то количество задач, выполняемых на включающем операцию `coalesce` этапе, эквивалентно количеству секций итогового набора RDD преобразования `coalesce`.

При использовании coalesce количество секций может уменьшиться на одном этапе даже без перетасовки. Однако операция coalesce приводит к тому, что расположенные далее по конвейеру секции всего этапа будут вычисляться с задаваемой coalesce степенью параллелизма, а это в некоторых случаях может оказаться нежелательным. Избежать такого поведения за счет перетасовки можно, задав аргументу shuffle функции coalesce значение true или применив вместо нее функцию repartition.

Однако, когда `coalesce` увеличивает количество секций, соблюдается правило: каждая из родительских секций обязательно зависит от нескольких дочерних. Следовательно, в соответствии с более точным определением широких зависимостей, приведенным в разделе «Узкие и широкие преобразования» на с. 108, использование функции `coalesce` для увеличения количества секций представляет собой широкое преобразование. Функция `coalesce` придает основное значение равномерному распределению данных по дочерним секциям. Соответственно, невозможно определить расположение записей в выходных данных во время проектирования, поскольку оно зависит от количества хранимых в каждой исходной секции записей, которое, конечно, нельзя определить без чтения данных и выполнения предыдущих преобразований. Таким образом, повышение количества секций с помощью вызова функции `coalesce` или функции `repartition` требует перетасовки.

Какой тип набора RDD возвращает преобразование

Наборы RDD — вдвойне абстрактное понятие: типы записей в них могут быть практически какими угодно (например, `String`, `Row`, `Tuple`) и относиться к одной из нескольких реализаций интерфейса `RDD`, обладающих различными свойствами. Оба этих отличия существенны для производительности и выполнения вычислений. Первое важно потому, что некоторые преобразования можно применять только к наборам RDD с определенным типом записей. Второе — потому, что каждое преобразование возвращает одну из нескольких реализаций интерфейса RDD, а значит, одно и то же преобразование, вызванное для двух различных реализаций RDD (например, `mappedRDD` и `GoGroupedRDD`), может вычисляться по-разному. В частности, отдельные реализации RDD сохраняют от предыдущих преобразований информацию об упорядоченности или локальности записей в RDD. А знания о локальности данных и секционировании итогового RDD-преобразования позволят избежать ненужных перетасовок. Более подробно этот вопрос изложен в подразделе «Сохранение информации о секционировании от преобразования к преобразованию» на с. 170, поскольку прежде всего относится к парам наборов RDD. В данном разделе мы обсудим сохранение информации о типе записей, так как это может быть важно для производительности и представляет собой на удивление непростую задачу в случае сложных программ Spark.

Набор RDD — тип-коллекция, экземпляры которого, аналогично типам коллекций в языках Scala, Java и большинстве сильно типизированных языков программирования, создаются в соответствии с параметром типа, указывающим тип членов коллекции.

В языке Scala синтаксис параметра типа — квадратные скобки, например List[String], символизирующие последовательность[1] объектов типа String. Это эквивалентно синтаксису < > языка Java (например, List<String>).

Типизация наборов RDD аналогична. Так, если вы используете метод `sc.textfile` для чтения набора RDD, то получите RDD данных типа `String` (обозначаемый `RDD[String]` в языке Scala и `RDD<String>` в языке Java).

Информация о типе записей набора RDD важна потому, что многие преобразования возможны только с RDD данных определенного типа, вследствие чего попытка воспользоваться этими методами для RDD обобщенного типа приведет к ошибкам во время компиляции или выполнения. Например, если была потеряна информация о типе для RDD кортежей и он был интерпретирован компилятором как `RDD[Any]` или даже `RDD[(Any, Int)]`, то вызов метода `sortByKey` не пройдет

[1] Точнее, список. — *Примеч. пер.*

компиляцию. Ошибка возникнет потому, что `sortByKey` можно вызывать только для наборов RDD пар «ключ — значение», в которых у ключей имеется какая-либо неявная упорядоченность. Аналогично числовые функции (например, `max`, `min` или `sum`) можно вызывать только для RDD данных типов `Long`, `Int` или `Double`.

Информация о типе записей — одно из многих мест в API Spark, где неявные преобразования типов могут привести к проблемам. При написании подпрограмм, предназначенных для использования в преобразованиях наборов RDD, зачастую будет лучше задать входной и возвращаемый типы вспомогательной функции явным образом и избегать указания их в виде обобщенного типа.

Один из случаев, когда потеря информации о типе часто приводит к проблемам, — работа с наборами `DataFrame` как RDD. Такие наборы могут неявно преобразовываться в RDD данных типа `Row`. Однако, поскольку объект `Row` модуля Spark SQL не является сильно типизированным (его можно создать из последовательности значений любого типа), компилятор Scala не способен «вспомнить» тип значения, использовавшегося для создания строки. Индексация последней возвращает значение типа `Any`, которое для выполнения большинства вычислений необходимо привести к более конкретному типу, например `String` или `Int`. Информация о типе строк хранится в схеме. Однако преобразование в тип RDD приводит к потере информации о схеме типа `DataFrame`, так что важно привязать схему `DataFrame` к переменной. Одно из преимуществ API Dataset заключается в его сильной типизации, поэтому значения в каждой из строк сохранят информацию о своем типе даже после преобразования последнего в RDD.

Минимизация количества создаваемых объектов

Сборка мусора — процесс освобождения выделенной под объект памяти после того, как этот объект становится не нужен. Поскольку фреймворк Spark работает в JVM, в которой имеется автоматическое управление памятью и большими структурами данных, то сборка может быстро превратиться в весьма затратную составляющую заданий Spark. Сборка мусора и происходящие во время нее ошибки — частая причина сбоев. Даже если накладные расходы сборки не настолько велики, чтобы задание вообще не могло работать, она все равно увеличивает время на сериализацию, а это замедляет выполнение задания. Минимизировать затраты на сборку мусора можно, снизив количество объектов и их размеры. Здесь будет уместно повторно использовать уже существующие объекты и структуру данных (например, простых типов данных), занимающие меньше места в памяти.

Повторное использование существующих объектов

Некоторые преобразования наборов RDD позволяют модифицировать параметры лямбда-выражений вместо возвращения нового объекта. Например, в функции/операции формирования последовательностей функции агрегирования для

`aggregateByKey` и `aggregate` можно модифицировать исходный аргумент-накопитель и описать функцию объединения таким образом, чтобы группа создавалась путем модификации первого из двух накопителей. Часто используемая и вполне эффективная парадигма выполнения сложных операций агрегирования состоит в описании класса языка Scala, содержащего операции формирования последовательностей и объединения, возвращающие существующий объект с помощью аннотаций `this.type`.

Например, нужно выполнить какую-то операцию агрегирования, не определенную во фреймворке Spark. Пусть у нас имеется набор RDD пар «ключ — значение», где ключи — панды-тренеры, а значения — табели успеваемости панд-учеников. Требуется получить для каждого тренера длину самого длинного из слов в табеле, среднее количество слов в одном табеле успеваемости и количество вхождений слова «счастлив». Один из удобных и понятных подходов — воспользоваться функцией `aggregateByKey`, принимающей три аргумента: нулевое значение, соответствующее пустому накопителю, функцию формирования последовательности, принимающую на входе накопитель и значение и добавляющую значение к накопителю, а также оператор объединения, определяющий, как необходимо объединить накопители. В данном случае можно сделать накопитель объектом с четырьмя полями: общее количество слов, общее количество табелей, самое длинное из найденных пока что слов и общее количество упоминаний слова «счастлив».

Для большей ясности можно описать это как отдельный объект с методами для формирования последовательностей и группировки. Мы назовем данный объект `MetricsCalculator`, его вероятный код показан в примере 5.2.

Пример 5.2. Пользовательский объект для операции агрегирования

```
class MetricsCalculator(
  val totalWords : Int,
  val longestWord: Int,
  val happyMentions : Int,
  val numberReportCards: Int) extends Serializable {

  def sequenceOp(reportCardContent : String) : MetricsCalculator = {
    val words = reportCardContent.split(" ")
    val tW = words.length
    val lW = words.map( w => w.length).max
    val hM = words.count(w => w.toLowerCase.equals("счастлив"))

    new MetricsCalculator(
      tW + totalWords,
      Math.max(longestWord, lW),
      hM + happyMentions,
      numberReportCards + 1)
  }

  def compOp(other : MetricsCalculator) : MetricsCalculator = {
    new MetricsCalculator(
      this.totalWords + other.totalWords,
```

```
      Math.max(this.longestWord, other.longestWord),
      this.happyMentions + other.happyMentions,
      this.numberReportCards + other.numberReportCards)
  }

  def toReportCardMetrics =
    ReportCardMetrics(
      longestWord,
      happyMentions,
      totalWords.toDouble/numberReportCards)
}
```

Далее можно воспользоваться этим объектом в аргументах нашей функции агрегирования, как показано в примере 5.4, в подпрограмме, которая выполняет отображение RDD тренеров и текста табелей в case-класс с тремя интересующими нас показателями и приведена в примере 5.3.

Пример 5.3. Case-класс для сводных показателей

```
case class ReportCardMetrics(
  longestWord : Int,
  happyMentions : Int,
  averageWords : Double)
```

Пример 5.4. Агрегирование без повторного использования объектов

```
/**
 * Агрегировать заданный набор RDD из (PandaInstructor, ReportCardText)
 * в RDD из неповторяющихся ключей
 * (PandaInstructor, ReportCardStatistics),
 * где ReportCardMetrics представляет собой case-класс
 *
 * longestWord -> Самое длинное из слов, содержащихся в написанных
 * тренером табелях успеваемости
 * happyMentions -> Количество упоминаний данным тренером слова «счастлив»
 * averageWords -> Среднее количество слов в табелях данного тренера
 */
def calculateReportCardStatistics(rdd : RDD[(String, String)]
): RDD[(String, ReportCardMetrics)] ={

  rdd.aggregateByKey(new MetricsCalculator(totalWords = 0,
    longestWord = 0, happyMentions = 0, numberReportCards = 0))(
    seqOp = ((reportCardMetrics, reportCardText) =>
      reportCardMetrics.sequenceOp(reportCardText)),
    combOp = (x, y) => x.compOp(y))
  .mapValues(_.toReportCardMetrics)
}
```

Данный метод лучше использования двух вызовов методов `map` и одного `reduceByKey`. Функция агрегирования объединяет все секции локально, после чего перетасовывает для выполнения межсекционной свертки. Однако у метода есть недостаток, заключающийся в создании нового экземпляра нашего пользовательского объекта для каждой записи набора данных и для каждого шага объединения.

Очень простой способ снизить затраты на создание объектов — модифицировать `ReportCardMetrics` для применения парадигмы проектирования `this.type` языка Scala так, чтобы операция формирования последовательностей модифицировала изначальный накопитель, а операция объединения модифицировала первый накопитель вместо возврата нового, как показано в примере 5.5.

Пример 5.5. Агрегирование с повторным использованием объектов

```
class MetricsCalculatorReuseObjects(
  var totalWords : Int,
  var longestWord: Int,
  var happyMentions : Int,
  var numberReportCards: Int) extends Serializable {

  def sequenceOp(reportCardContent : String) : this.type = {
    val words = reportCardContent.split(" ")
    totalWords += words.length
    longestWord = Math.max(longestWord, words.map( w => w.length).max)
    happyMentions += words.count(w => w.toLowerCase.equals("счастлив"))
    numberReportCards +=1
    this
  }

  def compOp(other : MetricsCalculatorReuseObjects) : this.type = {
    totalWords += other.totalWords
    longestWord = Math.max(this.longestWord, other.longestWord)
    happyMentions += other.happyMentions
    numberReportCards += other.numberReportCards
    this
  }

  def toReportCardMetrics =
    ReportCardMetrics(
      longestWord,
      happyMentions,
      totalWords.toDouble/numberReportCards)
}
```

Процедура агрегирования останется такой же.

Должно быть очевидно, что код Scala внутри операции формирования последовательностей работает медленнее, чем мог бы. Вместо трех различных вызовов функций с массивом words в качестве аргумента следовало бы воспользоваться строкой как строковым буфером, подсчитать слова, не забывая отслеживать самое длинное слово и подсчитывая вхождения слова «счастлив» (или по крайней мере применить цикл while для разбора массива words вместо трех рекурсивных вызовов). Мы оставили данный вариант решения потому, что он представляется более читабельным, а основная цель примера — продемонстрировать возможность оптимизации процедуры aggregateByKey фреймворка Spark.

Операции `reduce` (вызывающая операцию агрегирования) и `fold` (`foldLeft`, `fold`, `foldRight`) также извлекают выгоду из повторного использования объектов. Однако это единственные такие функции агрегирования. Лучше избегать изменяемых структур данных в коде Spark (да и вообще в коде на языке Scala), поскольку они могут приводить к ошибкам сериализации и неправильным результатам. Небезопасна модификация первого аргумента и для многих других функций наборов RDD, особенно узких преобразований, так как последние могут быть организованы цепочкой с применением отложенного вычисления, вследствие чего будут вычислены несколько раз. Например, при наличии набора RDD изменяемых объектов модификация массивов с помощью функции `map` может приводить к неправильным результатам в силу повторного использования объектов большее количество раз, чем ожидалось, особенно при повторном вычислении набора RDD.

Использование меньших структур данных

Фреймворк Spark способен оказаться «пожирателем» памяти. Один из важных способов оптимизации заданий Spark как в смысле памяти, так и времени выполнения — стараться задействовать простые типы данных, а не пользовательские классы. Хотя от этого и может пострадать удобочитаемость кода, применение массивов вместо case-классов или кортежей понижает затраты на сборку мусора. Массивы языка Scala, по сути являющиеся массивами языка Java, — наиболее эффективный по памяти тип коллекций Scala. Кортежи Scala представляют собой объекты, так что в некоторых случаях вместо кортежа для ресурсоемких операций предпочтительнее задействовать двух- или трехэлементный массив. Типы-коллекции языка Scala в целом приводят к бо́льшим накладным расходам, чем массивы.

Обратите внимание: наш объект `ReportCardMetrics` — просто адаптер для нескольких числовых значений. Несмотря на меньшую удобочитаемость и объектоориентированность, с точки зрения применяемого пространства эффективнее было бы воспользоваться четырехэлементным массивом целых чисел. Достичь того же уровня удобочитаемости кода можно с помощью ключевого слова `object` языка Scala вместо `class` с описанием операций формирования последовательности и объединения как функций над строками и массивами (пример 5.6).

Пример 5.6. Использование массива в качестве объекта для сводных показателей

```
object MetricsCalculator_Arrays extends Serializable {
  val totalWordIndex = 0
  val longestWordIndex = 1
  val happyMentionsIndex = 2
  val numberReportCardsIndex = 3

  def sequenceOp(reportCardMetrics : Array[Int],
    reportCardContent : String) : Array[Int] = {

    val words = reportCardContent.split(" ")
```

```
    // Модифицируем каждый из элементов массива
    reportCardMetrics(totalWordIndex) += words.length
    reportCardMetrics(longestWordIndex) = Math.max(
      reportCardMetrics(longestWordIndex),
      words.map(w => w.length).max)
    reportCardMetrics(happyMentionsIndex) += words.count(
      w => w.toLowerCase.equals("счастлив"))
    reportCardMetrics(numberReportCardsIndex) +=1
    reportCardMetrics
  }

  def compOp(x : Array[Int], y : Array[Int]) : Array[Int] = {
    // Объединяем первый и второй массивы путем модификации
    // элементов первого массива
    x(totalWordIndex) += y(totalWordIndex)
    x(longestWordIndex) = Math.max(x(longestWordIndex), y(longestWordIndex))
    x(happyMentionsIndex) += y(happyMentionsIndex)
    x(numberReportCardsIndex) += y(numberReportCardsIndex)
    x
  }

  def toReportCardMetrics(ar : Array[Int]) : ReportCardMetrics =
    ReportCardMetrics(
      ar(longestWordIndex),
      ar(happyMentionsIndex),
      ar(totalWordIndex)/ar(numberReportCardsIndex)
    )
}
```

Затем необходимо несколько изменить наш код для агрегирования. Мы уже не задействуем тот же пользовательский объект для сводных показателей, и нулевое значение поменялось. Все это показано в примере 5.7.

Пример 5.7. Агрегирование с применением массивов для минимизации ресурсоемкого создания объектов

```
def calculateReportCardStatisticsWithArrays(rdd : RDD[(String, String)]
): RDD[(String, ReportCardMetrics)] = {

  rdd.aggregateByKey(
    // Нулевое значение представляет собой четырехэлементный массив нулей
    Array.fill[Int](4)(0)
  )(
  // Операция seqOp добавляет в массив соответствующие значения
    seqOp = (reportCardMetrics, reportCardText) =>
      MetricsCalculator_Arrays.sequenceOp(reportCardMetrics, reportCardText),
  // Операция combo определяет способ объединения массивов
    combOp = (x, y) => MetricsCalculator_Arrays.compOp(x, y))
  .mapValues(MetricsCalculator_Arrays.toReportCardMetrics)
}
```

Часто бывает выгодно избегать создания внутри функции промежуточных объектов. Важно помнить, что преобразование между типами (например, различными видами коллекций языка Scala) приводит к созданию промежуточных объектов.

Это еще одно место, где неявные преобразования типов могут приводить к нежелательным последствиям для производительности.

Допустим, прочитав примечание из предыдущего раздела, вы захотели ускорить работу функции формирования последовательностей объекта `MetricsCalculator_ReuseObjects`. Далее вы обнаружили, что ваш сотрудник написал универсальную функцию для поиска вхождений слова «счастлив» и самого длинного слова в коллекции строк (пример 5.8).

Пример 5.8. Функция с неявными преобразованиями типов последовательностей

```
def findWordMetrics[T <:Seq[String]](collection : T ): (Int, Int)={
  val iterator = collection.toIterator
  var mentionsOfHappy = 0
  var longestWordSoFar = 0
  while(iterator.hasNext){
    val n = iterator.next()
    if(n.toLowerCase == "счастлив"){
      mentionsOfHappy +=1
    }
    val length = n.length
    if(length> longestWordSoFar) {
      longestWordSoFar = length
    }

  }
   (longestWordSoFar, mentionsOfHappy)
}
```

Этот сотрудник удачно определил данную функцию как работающую с любым типом данных, расширяющим индекс `Traversable` языка Scala. Следовательно, вообще не нужно преобразовывать тип массива слов и можно просто написать код, приведенный в примере 5.9.

Пример 5.9. Агрегирование с неудачными неявными преобразованиями типов

```
val totalWordIndex = 0
val longestWordIndex = 1
val happyMentionsIndex = 2
val numberReportCardsIndex = 3
def fasterSeqOp(reportCardMetrics : Array[Int], content   : String): Array[Int] = {
  val words: Seq[String] = content.split(" ")
  val (longestWord, happyMentions) = CollectionRoutines.findWordMetrics(words)
  reportCardMetrics(totalWordIndex) += words.length
  reportCardMetrics(longestWordIndex) = longestWord
  reportCardMetrics(happyMentionsIndex) += happyMentions
  reportCardMetrics(numberReportCardsIndex) +=1
  reportCardMetrics
}
```

К сожалению, в смысле создания объектов эта реализация еще хуже, чем предыдущая. Она создает два дополнительных объекта, содержащих коллекции со сло-

вами при каждом вызове операции формирования последовательностей! Сначала при вызове подпрограммы `findWordMetrics` в силу необходимости преобразовать входной массив в объект `Traversable` (при этом создается новый объект того же размера), а затем при приведении типа объекта `Traversable` к типу `Iterator`.

Изменять передаваемое в преобразование значение не всегда безопасно, так что внимательно изучите документацию по используемой для этого функции.

Помимо свертки объектов, под которые память выделяется непосредственно, неявные преобразования типов языка Scala иногда могут приводить к выделению дополнительной памяти в процессе преобразования типа.

Выполнение преобразований «итератор — итератор» с помощью функции mapPartitions

Функция `mapPartitions` набора RDD принимает на входе в качестве аргумента функцию, в которой один `iterator` по записям (представляющий записи из одной секции) переходит в другой (представляющий полученную секцию).

Преобразование `mapPartitions` — одно из наиболее мощных в Spark, ведь благодаря ему пользователь имеет возможность описать выполнение произвольной процедуры над одной из секций данных. Преобразование `mapPartitions` можно применять для очень простых преобразований данных, например синтаксического разбора строк. Однако оно подойдет и для очень сложной, ресурсоемкой обработки данных в целях решения таких задач, как вторичная сортировка или вычисление весьма нестандартных сводных показателей. Функция `mapPartitions` позволяет выполнять многие другие преобразования Spark, к примеру `filter`, `map` и `flatMap`. Оптимизация процедур `mapPartitions` — важная составляющая написания сложного и высокопроизводительного кода Spark, как мы увидим в главе 6. Фреймворку Spark нужно дать возможность сбрасывать часть записей на диск. Для этого важно делать функции внутри `mapPartitions` такими, чтобы они не требовали загрузки всей секции в оперативную память (допустим, из-за неявного преобразования ее в список). У итераторов имеется множество методов, подходящих для написания преобразований в стиле функционального программирования. Можно также создавать собственные пользовательские итераторы путем расширения интерфейса `Iterator`. Мы будем называть преобразования, которые принимают на входе итератор и его же возвращают без применения промежуточной коллекции, преобразованиями *«итератор — итератор»* (iterator-to-iterator).

Что такое преобразование «итератор — итератор»

Объект `iterator` языка Scala является на самом деле не коллекцией, а функцией, задающей процесс обращения к элементам коллекции по одному. Итераторы не только неизменяемы, но к одному элементу в итераторе можно обратиться только один раз. Другими словами, подразумевается только однократный проход итераторов — они выступают расширением интерфейса `TraversableOnce` языка Scala. Некоторые из определенных для итераторов методов аналогичны методам других неизменяемых коллекций, например методы отображения (`map` и `flatMap`), сложения (`++`), свертки (`foldLeft`, `reduceRight`, `reduce`), определения условий для элементов (`forall` и `exists`), а также обходов (`next` и `foreach`). В ряде случаев эти методы ведут себя не так, как у других коллекций языка Scala. А поскольку итератор можно пройти лишь один раз, то все методы итераторов, для которых требуется просмотр всех элементов, оставляют исходный итератор пустым.

В языке Java есть своя реализация итераторов, интерфейс java.util.Iterator, с теми же преимуществами для вычислений Spark, что и итераторы языка Scala.

Будьте осторожнее с вызовами функций. Нет ничего проще, чем случайно израсходовать итератор путем обращения к объекту, производящему обход итератора, например size, или вызвать срабатывание неявного преобразования типа. Итераторы можно преобразовать в любой другой тип коллекций языка Scala. Однако это требует обращения к каждому из их элементов. Следовательно, после преобразования в другой тип-коллекцию внутренний указатель исходного итератора будет находиться на последнем элементе (итератор будет пуст).

В известной мере полезно представлять себе методы итераторов так, как мы сделали это с методами наборов RDD: в качестве или *преобразования*, или *действия*, поскольку, подобно RDD, итераторы представляют собой скорее набор инструкций по вычислению, а не сохраненное состояние. Некоторые из методов итераторов, например `next`, `size` и `foreach`, выполняют обход итератора и его вычисление (что напоминает скорее действие). Другие, такие как `map` и `flatMap`, возвращают новый итератор — фактически представляющий набор инструкций по вычислению (подобно тому как преобразования наборов RDD возвращают новый такой набор). Однако, в отличие от преобразований Spark, преобразования итераторов совершаются линейно, по одному элементу за раз, а не параллельно. Это замедляет работу итераторов, но сильно упрощает их использование по сравнению с параллельным выполнением. Так, если необходимо сохранить некую информацию о просмотрен-

ных записях, то можно сделать это в функции `filter` или `map` итератора, поскольку операции отображения/фильтрации будут последовательно применены к каждому из элементов (пример 5.12 в конце данного раздела). Взаимно однозначные функции также не организовываются в цепочки в операциях над итераторами, вследствие чего три вызова функции `map` потребуют просмотра всех элементов итератора три раза.

Под преобразованием «итератор — итератор» мы понимаем использование одного из этих преобразований итераторов для получения нового итератора вместо: а) преобразования итератора в другой тип коллекций или б) вычисления итератора с помощью одного из «действий» итератора и создания новой коллекции. Повторим: использование цикла `while` для обхода элементов итератора и создания новой коллекции (даже нового итератора) *не является* преобразованием «итератор — итератор». Преобразование итератора в более интуитивно понятный тип коллекций, выполнение над ним каких-либо манипуляций и преобразование его обратно в итератор не есть преобразование «итератор — итератор». Разумеется, все выгоды от такого преобразования теряются при преобразовании типа аргумента итератора в функции `mapPartitions` в объект-коллекцию.

Преимущества в смысле занимаемого места и времени выполнения

Использование преобразований «итератор — итератор» в программах Spark выгодно прежде всего тем, что позволяет фреймворку избирательно сбрасывать данные на диск. По сути, это преобразование означает установку процесса вычисления элементов по одному. Следовательно, Spark может применить эту процедуру к группам записей, а не читать всю секцию в оперативную память или создавать в оперативной памяти коллекцию из всех выходных записей с последующим ее возвращением. Соответственно, благодаря преобразованиям «итератор — итератор» фреймворк Spark может работать с секциями, не помещающимися в оперативной памяти отдельного исполнителя.

Более того, представление секции в виде итератора обеспечивает фреймворку Spark возможность более избирательно использовать место на диске. Преобразования «итератор — итератор» позволяют вместо сброса на диск всей не помещающейся в оперативной памяти секции сбрасывать только не помещающиеся записи, а значит, экономить операции дискового ввода/вывода и затраты на повторное вычисление. Наконец, благодаря применению методов итераторов можно избежать описания промежуточных структур данных. Снижение количества больших промежуточных структур данных — один из способов обойтись без создания лишних объектов, которое приводит к замедлению сборки мусора (мы обсуждали это в разделе «Минимизация количества создаваемых объектов» выше).

К сожалению, API-функции mapPartitions библиотеки Spark Streaming — одно из относительно немногих мест, в которых API Scala решительно обходит по производительности своего конкурента из языка Java. До Spark 1.6 функция mapPartitions в Spark Streaming работала с объектами типа Iterable, а не Iterator языка Java и, следовательно, автоматически считывала в оперативную память всю коллекцию целиком. В Spark Core сгруппированный результат преобразования groupByKey в API Java все еще представляет собой тип Iterable, а не итератор, исключая, таким образом, возможность применения преобразования «итератор — итератор» после вызова groupByKey.

Пример

Несмотря на все указанные преимущества, итераторы — абстракция намного более сложная для осмысления и применения, чем типы коллекций, такие как массивы и хеш-карты, уже знакомые пользователям по другим языкам программирования. В этом разделе мы приведем пример сложной процедуры, задействующей `mapPartitions`, которая по заданному отсортированному набору RDD кортежей `((value, columnIndex), count)` и списку сводных показателей рейтинга по данной секции возвращает пары `(value, columnIndex)`, представляющие собой сводные показатели рейтинга (пример 5.10). Этот метод — часть оптимального решения задачи Златовласки, показанного полностью в подразделе «Златовласка. Версия 4: свертка до уникальных значений по каждой секции» на с. 193 и вкратце представленного в разделе «Пример со Златовлаской» на с. 151.

Пример 5.10. Вариант использования функции mapPartitions

```
private def findTargetRanksIteratively(
        sortedAggregatedValueColumnPairs : RDD[((Double, Int), Long)],
        ranksLocations : Array[(Int, List[(Int, Long)])]): RDD[(Int, Double)] = {

  sortedAggregatedValueColumnPairs.mapPartitionsWithIndex((partitionIndex : Int,
    aggregatedValueColumnPairs : Iterator[((Double, Int), Long)]) => {

    val targetsInThisPart: List[(Int, Long)] = ranksLocations(partitionIndex)._2
  if (targetsInThisPart.nonEmpty) {
    FindTargetsSubRoutine.asIteratorToIteratorTransformation(
      aggregatedValueColumnPairs,
      targetsInThisPart)
    } else {
      Iterator.empty
    }
  })
}
```

Эта процедура — хороший образец повышения производительности благодаря преобразованию «итератор — итератор», поскольку она выполняется по секциям, которые, похоже, окажутся слишком велики для того, чтобы поместиться в оперативной памяти. Однако в этом случае использование итераторов является с точки зрения проектирования неочевидным решением, ведь приходится вести карту про-

межуточных итогов с количеством элементов для каждого просмотренного до сих пор столбца. Возможна более простая архитектура данной процедуры: цикл по итератору, сохранение промежуточных итогов в объекте hashMap и создание новой коллекции необходимых элементов с помощью массива-буфера с последующим преобразованием последнего в итератор, как показано в примере 5.11.

Пример 5.11. Использование функции mapPartitions без преобразования «итератор — итератор»

```
def withArrayBuffer(valueColumnPairsIter : Iterator[((Double, Int), Long)],
  targetsInThisPart: List[(Int, Long)] ): Iterator[(Int, Double)] = {

    val columnsRelativeIndex: Predef.Map[Int, List[Long]] =
      targetsInThisPart.groupBy(_._1).mapValues(_.map(_._2))

  // Индексы столбцов пар с нужными сводными показателями,
  // расположенными в данной секции
    val columnsInThisPart: List[Int] = targetsInThisPart.map(_._1).distinct

  // Объект HashMap с промежуточными итогами каждого из индексов столбцов
  // Во время цикла по итератору мы будем обновлять хеш-карту
  // при обнаружении элементов каждого из индексов столбцов
    val runningTotals : mutable.HashMap[Int, Long]= new mutable.HashMap()
    runningTotals ++= columnsInThisPart.map(columnIndex => (columnIndex, 0L)).toMap

  // Воспользуемся массивом-буфером для создания итогового итератора
    val result: ArrayBuffer[(Int, Double)] =
    new scala.collection.mutable.ArrayBuffer()

    valueColumnPairsIter.foreach {
      case ((value, colIndex), count) =>

        if (columnsInThisPart contains colIndex) {

          val total = runningTotals(colIndex)
          // Рейтинги, содержащиеся в данном элементе входного итератора,
          // получаем с помощью фильтрации
          val ranksPresent = columnsRelativeIndex(colIndex)
            .filter(index => (index <= count + total) && (index > total))
          ranksPresent.foreach(r => result += ((colIndex, value)))
          // Обновляем промежуточные итоги
          runningTotals.update(colIndex, total + count)
      }
    }
  // Преобразование типа
  result.toIterator
}
```

На первый взгляд это решение кажется неплохим, так как предполагается, что возвращаемое количество элементов будет невелико и массивы-буферы — довольно эффективный способ построения коллекций языка Scala. Однако если входные данные очень велики по сравнению с размером кластера, то на данном этапе нас все равно будут преследовать сбои и ошибки нехватки памяти. Эффективнее было бы использовать преобразования «итератор — итератор». Мы можем переделать эту

подпрограмму в искомое преобразование, хотя процедура и не параллелизуема (требует ведения списка промежуточных итогов). Выполнить задуманное позволит то обстоятельство, что вычисление необходимой подпрограммы возможно для одного элемента итератора без какой-либо информации о других элементах. Итоговое решение применит функцию `filter` итераторов — для исключения элементов, не входящих в итоговые данные, — и функцию `flatMap` для формирования нового итератора из элементов итоговых секций, как показано в примере 5.12.

Пример 5.12. Использование функции mapPartitions с преобразованием «итератор — итератор»

```
def asIteratorToIteratorTransformation(
  valueColumnPairsIter : Iterator[((Double, Int), Long)],
  targetsInThisPart: List[(Int, Long)] ): Iterator[(Int, Double)] = {

  val columnsRelativeIndex = targetsInThisPart.groupBy(_._1).mapValues(_.map(_._2))
  val columnsInThisPart = targetsInThisPart.map(_._1).distinct

  val runningTotals : mutable.HashMap[Int, Long]= new mutable.HashMap()
    runningTotals ++= columnsInThisPart.map(columnIndex => (columnIndex, 0L)).toMap

  // Отфильтровываем пары, у которых нет содержащегося в данной части
  // индекса столбца
  val pairsWithRanksInThisPart = valueColumnPairsIter.filter{
    case (((value, colIndex), count)) =>
      columnsInThisPart contains colIndex
    }

  // Устанавливаем соответствие пар valueColumn списку пар
  // (colIndex, value), соответствующих одному из желаемых
  // сводных показателей по данной секции
  pairsWithRanksInThisPart.flatMap{

    case (((value, colIndex), count)) =>

        val total = runningTotals(colIndex)
        val ranksPresent: List[Long] = columnsRelativeIndex(colIndex)
                                       .filter(index => (index <= count + total)
                                         && (index > total))

        val nextElems: Iterator[(Int, Double)] =
          ranksPresent.map(r => (colIndex, value)).toIterator

        // Обновляем промежуточные итоги
        runningTotals.update(colIndex, total + count)
        nextElems
  }
}
```

Описанный подход позволяет избирательно сбрасывать функции данных на диск, работая с каждым элементом итератора индивидуально. Такая реализация экономит место на диске благодаря постепенному формированию результата вместо хранения нового типа-коллекции в оперативной памяти в виде массива-буфера. И экономит немного ресурсов при сборке мусора ввиду того, что в качестве промежуточного шага не создается массив-буфер.

При использовании объекта ArrayBuffer с целью формировать новую коллекцию для функции mapPartitions всегда можно (и, вероятно, окажется более эффективным) задействовать функцию map или flatMap итератора для постепенного добавления новых элементов.

Операции с множествами

Во фреймворке Spark есть масса различных операций с множествами, частью весьма ресурсоемких, причем поведение некоторых отличается от математических определений эквивалентных операций. В этом разделе мы постараемся показать, как использовать эти операции безопасно и эффективно.

Поскольку значения в наборах RDD неуникальны, то основное отличие от математических операций с множествами заключается в обработке повторяющихся значений. Так, операция `union` просто объединяет свои аргументы, поэтому размер результата операции `union` будет всегда равен размеру объединения двух наборов RDD. Определения операций `intersection` и `subtract` аналогичны их теоретико-множественным аналогам, но, так как во входных наборах RDD (в отличие от математических множеств) могут быть повторяющиеся значения, результаты способны удивить. Вычитание наборов RDD приводит к удалению из первого набора RDD всех элементов, ключ которых присутствует во втором. Следовательно, при вычитании возможно, что результат окажется меньше, чем размер первого набора минус размер второго, а это нарушает один из законов теории множеств.

Так, простой модульный тест из примера 5.13 пройдет успешно.

Пример 5.13. Операция вычитания

```
val a = Array(1, 2, 3, 4, 4, 4, 4)
val b = Array(3, 4)
val rddA = sc.parallelize(a)
val rddB = sc.parallelize(b)
val rddC = rddA.subtract(rddB)
assert(rddC.count() < rddA.count() - rddB.count())
```

В Spark операция `intersection` совместно группирует наборы-аргументы, используя их значения в качестве ключей и убирая те элементы, которые не присутствуют в обоих наборах. Соответственно, результат операции `intersection` не содержит повторяющихся значений. Хотя такое поведение для операции `intersection` вполне ожидаемо, применение нескольких операций к множествам с наборами RDD, содержащими повторяющиеся значения, может привести к неожиданным последствиям. Объединение двух наборов RDD из примера 5.13 представляет собой набор RDD, содержащий два элемента, `1` и `2`. Следовательно, как показывает модульный тест из примера 5.14, не всегда можно восстановить `rddA` с помощью объединения результатов операций пересечения и вычитания[1].

[1] Если A и B — множества, то $(A - B) \cup (B \cap A) = A$ всегда. В Spark это не так. При наличии в rddA или rddB повторяющихся или пересекающихся ключей $(rddA - rddB) \cup (rddB \cap rddA)$ является подмножеством rddA.

Пример 5.14. Операция пересечения

```
val a = Array(1, 2, 3, 4, 4, 4, 4)
val b = Array(3, 4)
val rddA = sc.parallelize(a)
val rddB = sc.parallelize(b)
val intersection = rddA.intersection(rddB)
val subtraction = rddA.subtract(rddB)
val union = intersection.union(subtraction)
assert(!rddA.collect().sorted.sameElements(union.collect().sorted))
```

Чтобы наборы RDD вели себя более похоже на множества, можно воспользоваться операцией distinct до вычисления каких-либо операций с множествами. Однако вызов данной операции приведет к перетасовке, если объект Partitioner неизвестен[1].

Снижение затрат на подготовительные работы

Некоторые операции требуют выполнения настройки для каждого экземпляра Spark или секции, скажем, создания подключения к базе данных или установки генератора случайных чисел. Для преобразований можно воспользоваться функцией `mapPartitions`, провести подготовительные работы для секции в функции `map`, после чего совершить желаемое преобразование с помощью итератора для данной секции. Мы проиллюстрируем это на образце генератора псевдослучайных чисел (пример 5.15).

Пример 5.15. С помощью транслирующей переменной создаем по генератору случайных чисел для каждой из секций

```
rdd.mapPartitions{itr =>
  // Однократно создаем генератор случайных чисел
  val r = new Random()
  itr.filter(x => r.nextInt(10) == 0)
}
```

Важно: не забывайте использовать преобразование «итератор — итератор» для избирательного сброса данных на диск, как обсуждалось в разделе «Выполнение преобразований "итератор — итератор" с помощью функции mapPartitions» на с. 121.

Кроме этого паттерна для снижения накладных расходов подготовительных работ при преобразованиях, есть еще один часто используемый паттерн: создание под-

[1] На самом деле перетасовка произойдет в любом случае, хотя если метод секционирования известен заранее, то все дубликаты будут убраны на шаге свертки, до перетасовки. — *Примеч. пер.*

ключения внутри действия для сохранения данных. Если ваша программа выводит данные, то можете задействовать тот же паттерн, что и с `mapPartitions`, только с `foreachPartition`.

Если подготовительные работы допускают сериализацию, то распространить объект можно с помощью транслирующей переменной. В противном случае уместно будет применить транслирующую переменную с сигнатурой `transient lazy val` (пример 5.17).

Разделяемые переменные

В фреймворке Spark имеется два вида разделяемых переменных: транслирующие переменные и накопители, каждая из которых может записываться только в одном контексте (драйвер или работник соответственно) и читаться в другом. Транслирующие переменные могут записываться в драйверной программе, а читаться на исполнителях, а накопители — наоборот.

Транслирующие переменные

Транслирующие переменные — способ распространения предназначенной только для чтения копии локального значения с драйвера на все машины вместо передачи новой копии с каждой задачей. Такие переменные могут показаться не особенно полезными, ведь можно просто захватить локальную переменную в замыкании для передачи данных от драйвера на рабочие узлы. Однако экономия от передачи лишь одной копии из расчета на работника (worker), по сравнению с отправкой по одной копии для каждой задачи, может оказаться колоссальной, особенно если одна и та же транслирующая переменная задействуется в нескольких преобразованиях. Два распространенных примера использования транслирующих переменных: а) трансляция небольшой таблицы для соединения; б) трансляция модели машинного обучения для формирования прогнозов на основе наших данных.

Транслирующая переменная создается с помощью вызова метода `broadcast` объекта `SparkContext`. Это приводит к распространению значения по работникам и возвращает адаптер, который можно использовать для доступа к этим значениям путем обращения к `value`, как показано в примерах 5.16 и 5.17. Если транслирующая переменная была создана с изменяемыми входными данными, то последние не должны меняться после ее создания, ведь уже существующие экземпляры Spark этих обновлений не заметят, в то время как новые работники увидят новое значение.

Пример 5.16. Трансляция объекта HashSet некорректных местоположений панд в целях их последующей фильтрации

```
val invalid = HashSet() ++ invalidPandas
val invalidBroadcast = sc.broadcast(invalid)
input.filter{panda => !invalidBroadcast.value.contains(panda.id)}
```

Пример 5.17. Создаем по генератору случайных чисел для каждого из работников

```
class LazyPrng {
  @transient lazy val r = new Random()
}
def customSampleBroadcast[T: ClassTag](sc: SparkContext, rdd: RDD[T]): RDD[T]= {
  val bcastprng = sc.broadcast(new LazyPrng())
  rdd.filter(x => bcastprng.value.r.nextInt(10) == 0)
}
```

Значение транслирующей переменной должно быть локальным и сериализуемым: никакие наборы RDD или другие распределенные структуры данных не допускаются.

Фреймворк Spark задействует транслирующие переменные для объектов конфигурации заданий Hadoop и больших блоков кода пользовательских функций на языке Python. Когда транслирующая переменная больше не нужна, ее можно явным образом удалить, вызвав для нее метод `unpersist()`.

Накопители

Накопители — второй из типов разделяемых переменных фреймворка Spark. Они позволяют накапливать на работниках побочную информацию от преобразования или действия с последующей передачей результата обратно драйверу. В соответствии с моделью выполнения Spark приращивает накопители только при запуске (скажем, действием) вычислений. Если последние выполняются несколько раз, то фреймворк каждый раз обновляет накопитель. Подобный многократный подсчет будет уместен для информации уровня процесса, такой как вычисление полного времени разбора записей. Однако он может крайне плохо подходить для информации, относящейся к обработке данных, например подсчету количества некорректных записей.

API накопителей фреймворка Spark был модифицирован для версии 2.0 — данные примеры адаптированы к API 2.X, хотя примеры для версии 1.X по-прежнему доступны в специальном репозитории (https://github.com/high-performance-spark/high-performance-spark-examples/blob/master/src/main/scala/com/high-performance-spark-examples/transformations/Accumulators.scala).

Накопители могут вести себя непредсказуемым образом. В их нынешнем состоянии их лучше использовать там, где возможный многократный подсчет является желаемым поведением.

Существует немало встроенных типов накопителей, упрощающих их создание в распространенных сценариях использования. Накопители не предназначены для

сбора больших объемов информации, так что при добавлении значительного количества элементов в коллекцию или присоединении к строке лучше задействовать отдельное действие вместо накопителя. Операцией по умолчанию для числовых накопителей выступает +, поэтому применим ее для вычисления суммарной пушистости всех панд, как показано в примере 5.18.

Пример 5.18. Вычисляем пушистость панд с помощью накопителей

```
def computeTotalFuzzyNess(sc: SparkContext, rdd: RDD[RawPanda]):
    (RDD[(String, Long)], Double) = {
  // Создаем поименованный накопитель для чисел типа double
  val acc = sc.doubleAccumulator("fuzzyNess")
  val transformed = rdd.map{x => acc.add(x.attributes(0)); (x.zip, x.id)}
  // Значение накопителя по-прежнему нулевое
  // Примечание: этот пример небезопасен, ведь преобразование
  // может вычисляться несколько раз
  transformed.count() // Инициируем выполнение вычислений
   (transformed, acc.value)
}
```

Кроме того, накопители поддерживают множество разнообразных типов данных для ассоциативных операций, но некоторые из них более проблемные, чем другие. Для использования накопителя другого типа необходимо реализовать интерфейс `AccumulatorV2[InputType, ValueType]`, описав методы `reset`, `copy`, `isZero`, `value`, `merge` и `add`. Вы должны реализовать логику класса для отслеживания накапливаемых значений. В целом достаточно одной `var`, возможно, двух. Помимо требуемого метода, переопределите метод `resetAndCopy`, чтобы повысить производительность в конкретных сценариях.

Обычно методы `reset` и `copy` применяются совместно в виде метода `resetAndCopy`, который можно реализовать эффективнее и исключить этап копирования (как делается в обоих примерах пользовательских накопителей — 5.19 и 5.20). Метод `reset` сбрасывает значение текущего накопителя в ноль, так что при вызове метода `isZero` будет возвращено `true`. Метод `copy` предназначен для создания копии заданного накопителя с тем же значением, что и у текущего. Он вызывается при копировании значения на работниках с целью избежать расходов (и потенциальной путаницы) копирования ранее накопленных значений на драйверы.

Параметры типа интерфейса `AccumulatorV2` задают тип накапливаемого значения (`add`) и итоговый возвращаемый тип (`value`). Что существенно, они не ограничивают и не задают тип, используемый для хранения самого накопителя. Для отслеживания значений в следующих примерах задействуется только одна переменная. Однако не ограничивайте себя ею одной. Во многих числовых накопителях Spark используется две `var`.

Сигнатура типа метода `merge` для API накопителей включает получение на входе того же базового типа `AccumulatorV2`. Поскольку типаж `AccumulatorV2` не устанавливает, каким образом работники должны отслеживать значения во время вычислений, то придется приводить тип получаемого накопителя к ожидаемому типу,

чтобы получить возможность обращаться к своим внутренним накапливающим полям. Простая реализация этого показана в примере 5.19.

Пример 5.19. Вычисляем максимальный идентификатор панды

```
def computeMaxFuzzyNess(sc: SparkContext, rdd: RDD[RawPanda]):
    (RDD[(String, Long)], Option[Double]) = {
  class MaxDoubleAccumulator extends AccumulatorV2[Double, Option[Double]] {
    // В этой переменной мы будем накапливать наше значение
    var currentVal: Option[Double] = None
    override def isZero = currentVal.isEmpty

    // Сбрасываем значение текущего накопителя в ноль – используется
    // при отправке значения по сети на рабочих узлах
    override def reset() = {
      currentVal = None
    }

    // Копируем текущий накопитель – на самом деле это используется
    // только в контексте копирования и сброса, но раз уж это часть
    // общедоступного API, то не будем рисковать.
    def copy() = {
      val newCopy = new MaxDoubleAccumulator()
      newCopy.currentVal = currentVal
      newCopy
    }

    // Переопределим функцию копирования и сброса для быстроты –
    // нет необходимости копировать значение, если мы все равно
    // собираемся тут же сбросить его в ноль. Особой разницы для
    // Option[Double] нет, но для чего-то вроде Array[X] разница
    // может быть колоссальной

    override def copyAndReset() = {
      new MaxDoubleAccumulator()
    }

    // Добавляем новое значение (обращение к которому происходит
    // на стороне работника)
    override def add(value: Double) = {
      currentVal = Some(
        // Если значение есть – сравниваем его с новым, в противном
        // случае просто сохраняем новое значение в качестве
        // текущего максимума
        currentVal.map(acc => Math.max(acc, value)).getOrElse(value))
    }

    override def merge(other: AccumulatorV2[Double, Option[Double]]) = {
      other match {
        case otherFuzzy: MaxDoubleAccumulator =>
          // Если параметр option накопителя other задан, то выполняем
          // его слияние с помощью обычной функции add
          // Если же нет, то не делаем ничего
          otherFuzzy.currentVal.foreach(value => add(value))
        case _ =>
```

```
          // Поток выполнения не должен попадать сюда никогда,
          // фреймворк Spark будет вызывать метод merge лишь
          // с правильным типом данных, но merge может вызвать
          // и кто-нибудь еще, так что генерируем исключение,
          // просто на всякий случай
          throw new Exception("Непредусмотренный вызов merge с неподдерживаемым
                              типом" + other)
      }
    }
    // Возвращаем накопленное значение
    override def value = currentVal
  }
  // Создаем новый пользовательский накопитель
  val acc = new MaxDoubleAccumulator()
  sc.register(acc)
  val transformed = rdd.map{x => acc.add(x.attributes(0)); (x.zip, x.id)}
  // Значение накопителя по-прежнему равно None.
  // Примечание: этот пример небезопасен, ведь преобразование
  // может вычисляться несколько раз
  transformed.count() // Инициируем выполнение вычислений
   (transformed, acc.value)
}
```

Для вышеприведенного примера по-прежнему требуется совпадение типов результата и накапливаемых данных. Если нужно собрать все различающиеся элементы, то придется, вероятно, собирать их в `HashSet`, и типы могут оказаться различными. Это проиллюстрировано в примере 5.20.

Пример 5.20. Вычисляем уникальные идентификаторы панд

```
def uniquePandas(sc: SparkContext, rdd: RDD[RawPanda]): HashSet[Long] = {
  class UniqParam extends AccumulatorV2[Long, HashSet[Long]] {
    var accValue: HashSet[Long] = new HashSet[Long]()

    def value = accValue

    override def copy() = {
      val newCopy = new UniqParam()
      newCopy.accValue = accValue.clone
      newCopy
    }
    override def reset() = {
      this.accValue = new HashSet[Long]()
    }
    override def isZero() = {
      accValue.isEmpty
    }

    // Переопределим функцию копирования и сброса для быстроты —
    // нет необходимости копировать значение, если мы все равно
    // собираемся тут же его обнулить
    override def copyAndReset() = {
      new UniqParam()
    }
    // Для добавления новых значений
```

```
    override def add(value: Long) = {
      accValue += value
    }
    // Для слияния накопителей
    override def merge(other: AccumulatorV2[Long, HashSet[Long]]) = {
      other match {
        case otherUniq: UniqParam =>
          accValue = accValue ++ otherUniq.accValue
        case _ =>
          throw new Exception("Поддерживается слияние значений только
                              с одинаковым типом")
      }
    }
  }
  // Создаем накопитель для отслеживания уникальных значений
  val acc = new UniqParam()
  // Регистрируем, задавая название
  sc.register(acc, "Уникальные значения")
  val transformed = rdd.map{x => acc.add(x.id); (x.zip, x.id)}
  // Значение накопителя по-прежнему Double.MinValue.
  transformed.count() // Инициируем выполнение вычислений
  acc.value
}
```

Функция value может решать сложные задачи и возвращать тип данных, отличающихся от входного типа или внутреннего накапливаемого значения. Например, при вычислении среднего значения она способна делить одно длинное целое на другое и возвращать число типа double.

В языке Scala можно задавать названия накопителей, которые будут отображаться в пользовательском веб-интерфейсе. Достаточно просто добавить название в качестве второго параметра. Однако это приводит, конечно, к вызову для накопителя toString, поэтому если данная операция требует в вашем случае больших затрат ресурсов, то лучше оставить накопитель без названия.

При работе с кэшированными данными может показаться, что накопители работают практически согласованно, но, как будет видно из обсуждения в подразделе «Взаимодействие с накопителями» на с. 148, это не так.

Предлагается добавить накопители свойств данных («согласованные накопители») в версии 2.1 Spark[1]. Накопители свойств позволили бы избежать двойного подсчета — но пока еще этот механизм не включен во фреймворк. Вы можете следить за состоянием дел по следующему запросу на включение изменений (https://github.com/apache/spark/pull/11105).

[1] Изначально планировалось сделать это в версии 2.0.

Начиная с версии 2.0, Spark использует накопители для отслеживания показателей выполнения задач.

Повторное использование наборов RDD

Spark предлагает несколько вариантов повторного использования наборов RDD, включая сохранение, кэширование и создание контрольных точек. Однако фреймворк не делает ничего из перечисленного автоматически[1], поскольку сохранение RDD для повторного использования нарушает работу ряда конвейеров — пустая трата усилий в случае однократного применения этого RDD или если повторное вычисление преобразования не требует больших вычислительных затрат. Все виды сохранения (один из которых — кэширование) и создания контрольных точек означают определенные вычислительные затраты и вряд ли повысят производительность однократных операций. Более того, в случае больших наборов данных затраты на сохранение или создание контрольных точек могут оказаться настолько высоки, что лучшим выходом было бы повторное вычисление. Однако для некоторых особых видов программ Spark повторное использование набора RDD может привести к колоссальному выигрышу в производительности как в смысле повышения скорости работы, так и в области снижения частоты отказов.

Сценарии повторного использования

В этом разделе мы рассмотрим случаи, в которых сохранение или создание контрольных точек влечет повышение производительности. Вообще говоря, важнейшие сценарии повторного использования — применение набора RDD несколько раз, выполнение нескольких действий над одним набором, а также длинные (или требующие больших вычислительных затрат) цепочки преобразований.

Итеративные вычисления

Повторное использование RDD для преобразований, в которых один родительский набор RDD применяется несколько раз, позволяет принудительно инициировать совершение вычислений и таким образом избежать их повтора. Например, сохранение набора данных, с которым выполняется цикл соединений, может привести к колоссальному повышению производительности, поскольку гарантирует наличие секций данного набора RDD в оперативной памяти при всех соединениях.

В примере 5.21 вычисляется среднеквадратическая ошибка (root mean squared error, RMSE) по нескольким различным наборам RDD, представляющим прогнозы на

[1] Среди достойных упоминания исключений можно назвать некоторые алгоритмы ML, которые при передаче им несохраняемого набора RDD автоматически помещают его в хранилище, а потом удаляют оттуда.

основе разных моделей. Для этого необходимо соединить каждый из наборов RDD с прогнозами с набором проверочных данных.

В этом примере мы задействуем метод persist(), сохраняющий набор RDD в памяти. Как будет описано в подразделе «Виды повторного использования: кэш, сохранение, контрольная точка, перетасовочные файлы» на с. 140, вызов cache() эквивалентен вызову persist(), который, в свою очередь, эквивалентен вызову persist("MEMORY_ONLY").

Пример 5.21. Функция с итеративными вычислениями

```
val testSet: Array[RDD[(Double, Int)]] =
  Array(
    validationSet.mapValues(_ + 1),
    validationSet.mapValues(_ + 2),
    validationSet)
validationSet.persist()
// Сохраняем, поскольку используем данный набор RDD несколько раз
val errors = testSet.map( rdd => {
    rmse(rdd.join(validationSet).values)
})
```

Без сохранения фреймворку Spark пришлось бы перезагрузить и заново секционировать обучающий набор данных для завершения соединения. Однако благодаря сохранению обучающий набор RDD остается загруженным в оперативную память на исполнителях после каждого выполнения алгоритма. Подробнее мы обсудили вопросы производительности при различных типах соединений в разделе «Соединения Spark Core» на с. 95.

Создание контрольных точек — еще одна форма повторного использования наборов RDD, при которой RDD записывается во внешнее хранилище и происходит разбиение графа происхождения наборов. Однако благодаря созданию контрольных точек секции также остаются загруженными на исполнителях.

Несколько действий над одним набором RDD

Если не использовать повторно RDD, то каждое вызываемое для него действие приведет к запуску отдельного задания Spark с полной «родословной» преобразований набора RDD. Сохранение и создание контрольных точек разбивают граф происхождения наборов, так что одни последовательности преобразований перед вызовом `persist` или `checkpoint` будут вызываться лишь однократно. Поскольку сохраненный RDD и созданные контрольные точки существуют только на время жизни приложения Spark (хотя и *могут* вытесняться последующими сохраненными/закэшированными данными), то сохраненный во время одного задания Spark набор RDD будет доступен в следующем задании в рамках того же `SparkContext`. Допустим, требуется собрать первые 10 % записей набора RDD. Для этого мож-

но задействовать код из примера 5.22, который вызывает метод `sortByKey`, после чего — метод `count` и, наконец, `take`.

Пример 5.22. Два действия без этапа сохранения

```
val sorted = rddA.sortByKey()
val count = sorted.count()
val sample: Long = count / 10
sorted.take(sample.toInt)
```

Метод `sortByKey` (и, предположительно, операция чтения), необходимый для создания набора RDD `sorted`, будет выполняться дважды, если мы не сохраним RDD: один раз в задании, вызываемом методом `count`, а второй — в задании, вызываемом методом `take`. Мы не можем протестировать этот фрагмент выполнения программно, но, запустив данное приложение и заглянув в веб-интерфейс, увидим, что код запускает два задания, каждое из которых включает этап сортировки. Однако если добавить вызов операции сохранения или создания контрольной точки перед действиями (как показано в примере 5.23), то указанное преобразование будет выполняться только один раз. Так происходит потому, что фреймворк Spark строит *граф происхождения* (lineage graph), начиная или с создания набора RDD, или с момента сохранения либо создания контрольной точки для RDD.

Пример 5.23. Два действия с этапом сохранения

```
val sorted = rddA.sortByKey()
rddA.persist()[1]
val count = sorted.count()
val sample: Long = count / 10
sorted.take(sample.toInt)
```

Время жизни сохраненных наборов RDD не превышает времени жизни приложения Spark. Для повторного использования данных в других приложениях Spark примените создание контрольных точек в том же каталоге.

Если затраты на вычисление секций слишком высоки

Даже при условии, что один и тот же набор RDD не используется в программе несколько раз, сохранение и создание контрольной точки может ускорить выполнение процедуры и снизить стоимость отказа благодаря сохранению промежуточных результатов. Особенно полезно сохранять или создавать контрольные точки в случае крупных вычислительных затрат на вычисление одной секции, поскольку это гарантирует, что не нужно будет повторять всю дорогостоящую операцию при отказах далее по конвейеру.

[1] См. комментарий автора в списке замеченных неточностей на сайте издательства O'Reilly. — *Примеч. пер.*

Например, при необходимости выполнить в программе длинную последовательность взаимно однозначных преобразований они будут объединены в задачи, требующие чрезвычайно большого объема вычислений. Данное обстоятельство не вызывает проблем, если эти задачи решаются успешно и помещаются в оперативной памяти. Но в случае сбоя одного из расположенных далее по конвейеру преобразований затраты на повторное вычисление одной-единственной секции могут оказаться огромными. Особенно полезны создание контрольных точек или сохранение вне кучи, когда накладные расходы на сборку мусора или нагрузка на память всех узких преобразований, взятых вместе, превышают возможности исполнителей кластера. Как создание контрольных точек, так и сохранение вне кучи позволяют сохранить набор RDD вне памяти исполнителей Spark, оставляя свободную память для проведения вычислений. Кроме того, это единственные способы предотвращения повторных вычислений в случае полного отказа одного из работников Spark. Иногда само по себе разбиение длинного графа происхождения может обеспечить успешное выполнение задания, поскольку размер каждой из задач будет меньше.

Узкие преобразования обычно выполняются быстрее, чем широкие. Однако отдельные узкие преобразования, например посекционное обучение модели или работа с очень длинными строками, могут потребовать больших вычислительных затрат. В этих случаях повторное использование набора RDD после «дорогостоящих» вычислений с целью избежать вероятного повторного вычисления способно повысить производительность.

Как определить, достаточно ли низки затраты на повторное вычисление

Хотя сохранение в оперативной памяти — одна из важнейших возможностей Spark, оно требует определенных затрат. В частности, значительного места для хранения данных в оперативной памяти, а также времени на сериализацию и десериализацию. Как мы обсудим в подразделе «Разделение места внутри исполнителя» на с. 310, сохранение в памяти и выполняемые в ней вычисления осуществляются в JVM исполнителя Spark. Следовательно, сохраненные в оперативной памяти данные могут занимать место, которое в противном случае пригодилось бы для дальнейших вычислений, или приводить к повышению частоты ошибок памяти. Кэширование с помощью структур памяти языка Java (все предоставляемые Spark варианты, за исключением сохранения вне кучи) значительно повышает затраты на сборку мусора по сравнению с повторным вычислением.

Сохранению на диск или *созданию контрольных точек* (checkpointing, запись набора RDD во внешнюю файловую систему) присущи недостатки MapReduce вследствие высоких затрат на операции чтения и записи. При сохранении на диск или создании контрольной точки для набора RDD необходимо учитывать не только

дисковое пространство на кластере, требуемое для записи RDD, но и вычислительные затраты исполнителей Spark на дополнительные операции дискового ввода/вывода. В большинстве случаев создание контрольной точки для большого набора RDD применяется с целью снизить частоту отказов в кластерах с интенсивным обменом информацией, но редко приводит к повышению производительности, даже если набор приходится вычислять повторно, по причине высоких затрат на создание такой точки.

Наш опыт показывает: нет ничего проще, чем недооценить то, насколько затраты на хранение и чтение RDD велики по отношению к повторному вычислению. Мы также обнаружили, что для относительно простых операций затраты на чтение при загрузке набора RDD намного превышают остальные, поэтому сохранение полезнее всего, когда может предотвратить дополнительную операцию чтения или в случае нескольких итеративных вычислений.

Более того, разбиение графа происхождения набора RDD путем принудительного инициирования вычисления с помощью сохранения или создания контрольной точки предотвращает объединение преобразований с узкими зависимостями в единую задачу. Следовательно, мы теряем некоторые из узких преобразований, которые не получается объединить и выполнить в одной задаче. Например, выполнение сохранения или создание контрольной точки между шагами `map` и `filter` приведет к разрыву конвейера для сохранения некогда промежуточных данных. Вследствие этого Spark будет вынужден выполнить два прохода по данным вместо одного, поскольку для материализации набора RDD после операции `map` необходимо вычислить преобразование. Разбивать граф происхождения между узкими преобразованиями желательно лишь в самых исключительных случаях.

Предыдущие указания представляют собой неплохие эвристические правила для определения того, принесет ли повторное использование существенную выгоду. В целом повторно применять набор RDD, а не вычислять его заново, имеет смысл, когда объем вычислений достаточно велик относительно кластера и оставшейся части задания. Наилучший способ определить, следует ли повторно использовать RDD, — запустить выполнение задания. Если процесс идет очень медленно, стоит взглянуть, не поможет ли сохранение набора RDD, прежде чем пытаться переписать программу, ведь сохранение и создание контрольной точки способно снизить затраты на повторное вычисление данных в случае отказа или вообще их устранить. При сбое задания из-за ошибок сборки мусора или недостатка памяти создание контрольной точки или сохранение вне кучи может позволить завершить задание, особенно когда кластер зашумлен. С другой стороны, если вы уже используете сохранение, причем в оперативной памяти, то имеет смысл задуматься об удалении вызова операции сохранения или переходе на создание контрольной точки или сохранение/создание контрольной точки вне кучи.

При тестировании кода перед его вводом в эксплуатацию стоит параметризовать уровень сохранения переменной, через которую можно было задавать уровень сохранения с помощью аргумента командной строки. Эта идея используется в функции из примера 5.24, содержащей аргумент storageLevel (значение которого может равняться NONE).

Виды повторного использования: кэш, сохранение, контрольная точка, перетасовочные файлы

Фреймворк Spark предоставляет множество вариантов сохранения набора RDD в случае необходимости его повторного использования. Важно четко понимать, в каких случаях следует применять те или иные типы сохранения. Есть три основные операции, пригодные для хранения RDD: кэширование, сохранение и создание контрольной точки. В целом кэширование (эквивалент сохранению с помощью хранилища в оперативной памяти) и сохранение наиболее удобны для исключения повторного вычисления в одном задании Spark или для разбиения наборов RDD с длинными графами происхождения, поскольку наборы во время выполнения задания Spark в этом случае хранятся в исполнителях. Создание контрольных точек удобнее использовать в целях предотвращения отказов и больших затрат на повторные вычисления благодаря сохранению промежуточных результатов. Подобно сохранению, создание контрольных точек помогает исключить надобность в лишних вычислениях, минимизируя таким образом стоимость отказа, а также позволяет избежать повторного вычисления с помощью разбиения графа происхождения.

Сохранение и кэширование

Сохранение (persisting) набора RDD означает материализацию RDD (обычно путем размещения его в оперативной памяти) для повторного использования *во время текущего задания*. Spark запоминает граф происхождения сохраненного набора RDD и может его вычислить заново во время работы текущего задания, если одна из сохраненных секций будет утрачена. После окончания задания функция `persist` принимает на входе аргумент `StorageLevel`, указывающий, как необходимо сохранить набор. Фреймворк Spark предоставляет несколько констант для различных уровней сохранения, каждый из которых создается на основе пяти атрибутов хранения наборов RDD: `useDisk`, `useMemory`, `useOfHeap`, `deserialized` и `replication`. Параметры для уровня хранения можно узнать, вызвав для него метод `toString`. Посвященная сохранению документация Spark (http://spark.apache.org/docs/latest/rdd-programming-guide.html) включает весьма полный список доступных для применения вариантов хранения.

Тем не менее мы считаем, что не помешает привести чуть больше информации о каждом из пяти свойств, определяющих варианты хранения. Это позволит лучше понимать, какой из вариантов следует выбрать.

- `useDisk` — при указании данного свойства секции, не помещающиеся в оперативной памяти, будут записываться на диск.

  Это свойство активизируют содержащие `DISK` (например, `MEMORY_AND_DISK`) флаги для уровня хранения. По умолчанию не помещающиеся в оперативной памяти секции просто вытесняются, и их приходится вычислять повторно при использовании сохраненного набора RDD (см. подраздел «LRU-кэширование» на с. 145). Следовательно, сохранение на диск позволяет гарантированно избежать повторного вычисления этих дополнительных больших секций. Однако чтение с диска может потребовать значительного времени, так что сохранять на него имеет смысл только в случае относительно высокой стоимости повторного вычисления.

Разрешить запись на диск можно в случае предположения, что набор RDD не поместится в оперативной памяти. Однако если стоимость повторного вычисления секций не слишком высока (оно представляет собой обычные отображения, а не уменьшение размера данных), то быстрее будет вместо чтения с диска вычислить некоторые секции повторно.

- `useMemory` — при указании этого свойства набор RDD будет сохранен в оперативной памяти или непосредственно записан на диск.

  Единственные уровни хранения, в которых его значение равно `false`, — `DISK_ONLY`. Большая часть обеспечиваемого кэшированием ускорения работы происходит за счет хранения наборов RDD в оперативной памяти, так что если главная причина для повторного использования — обеспечение быстрого доступа для повторяемых вычислений, то можно выбрать именно такой вариант хранения, при котором секции хранятся в оперативной памяти. Однако в ряде ситуаций имеет смысл и сохранение наборов RDD только на диске, например, если вычисления требуют бóльших затрат, чем чтение с локального диска, или при слишком медленной работе сетевой файловой системы (скажем, в случае некоторых объектных хранилищ).

- `useOfHeap` — при указании этого свойства набор RDD будет храниться вне исполнителя Spark во внешней системе, например Tachyon.

  Это свойство активизируется параметром хранения `off_heap`. Оно может оказаться уместно, если нехватка памяти представляет собой серьезную проблему или кластер зашумлен и секции вытесняются. Мы обсудим преимущества использования Tachyon в подразделе «Alluxio (бывшая Tachyon)» на с. 144.

- `deserialized` — при указании этого свойства набор RDD будет храниться в виде десериализованных Java-объектов.

  Как мы обсудим в подразделе «Kryo» на с. 317, при этом наборы RDD занимают меньше места, особенно в случае использования быстрого сериализатора, но данное сочетание влечет определенные издержки. Сериализацию активизируют те параметры хранения, название которых включает суффикс `_SER`, например `MEMORY_ONLY_SER`.

Если набор RDD слишком велик для его сохранения в оперативной памяти, то попробуйте сначала сериализовать его с помощью параметра MEMORY_ONLY_SER. При этом скорость доступа к RDD не пострадает, а необходимый для его хранения объем памяти сократится.

- `replication` — представляет собой целочисленное значение, позволяющее управлять количеством хранимых в кластере копий сохраненных данных.

  По умолчанию оно равно `1`; однако заканчивающиеся на `_2` параметры сериализации, например `DISK_ONLY_2`, реплицируют каждую секцию по двум узлам. Этим параметром можно воспользоваться для ускорения работы механизма отказоустойчивости. Однако не забывайте, что сохранение с репликацией требует вдвое большего места на диске и вполовину замедляет сохранение по сравнению с сохранением без репликации. Репликация обычно требуется только в случае зашумленного кластера или плохого соединения, когда вероятность отказов сильно повышается. Она также может оказаться полезной, если у вас нет времени выполнять повторные вычисления в ситуации отказа, скажем, при обслуживании работающего онлайн веб-приложения.

RDD-операция cache() эквивалентна операции сохранения без задающего уровень хранения аргумента, то есть persist(). Как cache(), так и persist() сохраняют набор RDD с уровнем хранения по умолчанию: MEMORY_ONLY, эквивалентным вызову StorageLevel(false, true, false, true). Он сохраняет наборы в оперативной памяти в виде десериализованных Java-объектов, не записывая данные на диск при вытеснении секций и не реплицируя секции.

Создание контрольных точек

При создании контрольных точек набор RDD записывается во внешнюю систему хранения, например HDFS или S3, и — в отличие от сохранения — Spark «забывает» граф происхождения набора. Поскольку создание контрольных точек приводит к записи набора RDD вне Spark, а значит, инициации вычисления набора, то время жизни информации из контрольных точек превосходит время жизни отдельного приложения Spark. Создание контрольных точек требует больше места во внешнем хранилище и выполняется медленнее, чем сохранение, из-за необходимости совершать потенциально дорогостоящие операции записи. Однако оно не использует память Spark и не влечет повторных вычислений в случае сбоя работника Spark.

Рисунок 5.2 демонстрирует различие между сохранением в оперативной памяти и созданием контрольных точек для RDD. Сохранение приводит к размещению секций RDD в оперативной памяти или на диске в слое кэширования исполнителей. Создание контрольных точек влечет запись всех секций в какую-либо внешнюю систему.

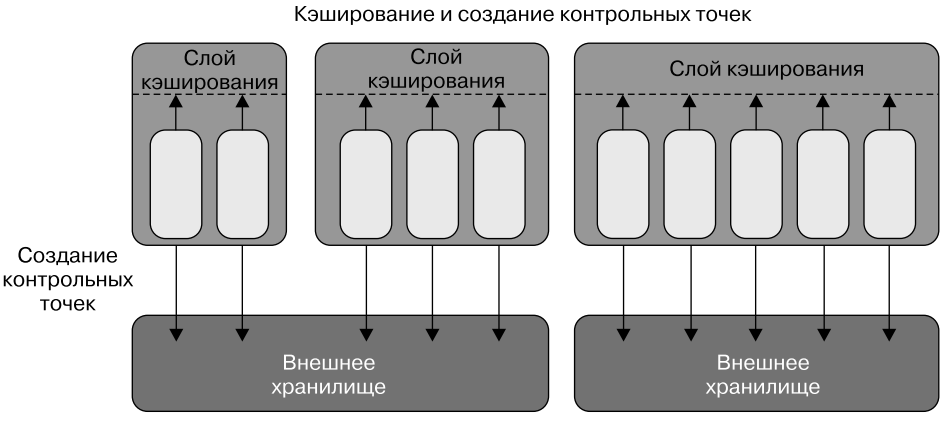

Рис. 5.2. Кэширование по сравнению с созданием контрольных точек

Лучше всего использовать кэширование, когда возможность отказа и повторного вычисления значит больше, чем дополнительное место во внешнем хранилище. Вообще говоря, мы рекомендуем применять сохранение в случае медленного выполнения заданий, а создание контрольных точек — когда они не выполняются вообще из-за сбоев. При сбое задания Spark из-за ошибок нехватки памяти создание контрольных точек снизит стоимость отказа и его вероятность без перерасхода памяти на исполнителях. Если же сбой происходит вследствие сетевых ошибок или их вытеснения в зашумленном кластере, то создание контрольных точек может снизить вероятность отказа с помощью разбиения длительного задания на меньшие сегменты. Чтобы вызвать метод `checkpoint`, сначала вызовите метод `setCheckpointDir(directory: String)` объекта `SparkContext` и передайте в него путь, по которому располагается HDFS, для записи промежуточных результатов. Затем в задании Spark вызовите метод `checkpoint()` для набора RDD.

Пример создания контрольной точки

В примере 5.24 мы задаем пользовательские уровень хранения и параметры создания контрольной точки. Эта функция используется в примере со Златовлаской, который мы опишем подробнее в подразделе «Златовласка. Версия 4: свертка до уникальных значений по каждой секции» на с. 193, и задействуем там пользовательский уровень хранения и параметры создания контрольной точки. В данном случае мы выполняем несколько весьма дорогостоящих преобразований: сначала сортировку, а затем две весьма основательные процедуры отображения секций. При работе на зашумленном кластере предпочтительно создать контрольную точку в этой функции после сортировки. Значение параметра `directory` — каталог для контрольных точек. Значение `sorted` — отсортированный набор RDD пар «ключ — значение».

Пример 5.24. Создание контрольной точки

```
def findQuantilesWithCustomStorage(valPairs: RDD[((Double, Int), Long)],
colIndexList: List[Int], targetRanks: List[Long], storageLevel: StorageLevel =
StorageLevel.MEMORY_AND_DISK, checkPoint : Boolean, directory : String = ""):
Map[Int, Iterable[Double]] = {
  val n = colIndexList.last + 1
  val sorted = valPairs.sortByKey()
  if (storageLevel != StorageLevel.NONE) {
    sorted.persist(storageLevel)
  }

  if (checkPoint) {
    sorted.sparkContext.setCheckpointDir(directory)
    sorted.checkpoint()
  }

  val partitionColumnsFreq = getColumnsFreqPerPartition(sorted, n)
  val ranksLocations = getRanksLocationsWithinEachPart(
    targetRanks, partitionColumnsFreq, n)
  val targetRanksValues = findTargetRanksIteratively(sorted, ranksLocations)
  targetRanksValues.groupByKey().collectAsMap()
}
```

Фреймворк Spark включает функцию локального создания контрольных точек, приводящую к усечению графа происхождения набора RDD без сохранения в устойчивое хранилище. Она не подходит для кластеров, в которых возможны отказы, вытеснение или динамическое понижающее вертикальное масштабирование в тот период времени, когда могут выполняться обращения к набору RDD.

Alluxio (бывшая Tachyon)

Tachyon — распределенная система хранения данных в оперативной памяти, разработанная отдельно от Spark. Она используется в качестве надстройки над какой-либо системой хранения, например S3 или HDFS, и может применяться самостоятельно или вместе с внешней вычислительной платформой, скажем Spark или MapReduce. Подобно фреймворку Spark, Tachyon способна работать в автономном режиме кластера или вместе с Mesos или YARN. Прочесть больше об архитектуре Tachyon и интеграции ее со Spark можно в тематической документации (https://www.alluxio.org/).

Tachyon можно применять как входной или выходной источник данных для приложений Spark (хранимые в Tachyon данные пригодны для создания наборов RDD) или для сохранения вне кучи (`off_heap`) во время работы приложения Spark. Использование Tachyon для сохранения имеет несколько преимуществ. Во-первых, оно снижает накладные расходы на сборку мусора, поскольку данные не хранятся в виде Java-объектов. Во-вторых, несколько исполнителей могут совместно задействовать в Tachyon один и тот же внешний пул памяти. В-третьих, поскольку данные хранятся в оперативной памяти вне Spark, они не будут утрачены в случае

фатальных сбоев отдельных исполнителей. Это особенно удобно при необходимости повторно использовать набор RDD в условиях недостатка оперативной памяти или ошибок сборки мусора. Это также оптимальный способ повторного применения очень большого набора RDD в нескольких приложениях.

Позиции разработчиков Tachyon и сообщества пользователей особенно сильны в Китае, так что часть ее документации может оказаться более полной на китайском языке, чем на английском.

LRU-кэширование

После того как сохраненные в оперативной памяти или на диске наборы RDD становятся более не нужны (для последующего использования далее по конвейеру), они не удаляются автоматически из хранилища. Вместо этого они остаются в памяти на всем протяжении жизни приложения Spark до тех пор, пока драйверная программа не вызовет функцию `unpersist` или потребности в памяти/хранилище не приведут к их вытеснению. Spark применяет кэширование типа *«наиболее давно использовавшиеся»* (least recently used, LRU) для определения того, какие секции необходимо вытеснить, если в исполнителях начнет заканчиваться память.

LRU-кэширование требует вытеснения структуры данных, к которой последний раз обращались наиболее давно. Однако в силу отложенного вычисления может оказаться непросто предсказать, какие секции окажутся вытеснены первыми. В целом Spark вытесняет самые старые секции, созданные или использовавшиеся в самых ранних заданиях или на самом раннем этапе в пределах конкретного задания (см. более подробные пояснения по поводу управления памятью и секций в подразделе «Разделение места внутри исполнителя» на с. 310). LRU-кэширование работает по-разному для различных опций сохранения. При операциях сохранения только в оперативной памяти, настроенных на использование LRU-кэширования, Spark будет повторно вычислять вытесненные секции при каждой необходимости. В случае сохранения в оперативной памяти и на диске LRU-кэширование приведет к записи вытесненных секций на диск. При необходимости удалить сохраненный набор RDD из оперативной памяти, чтобы освободить место, можно задействовать функцию `unpersist`.

Перетасовочные файлы. Независимо от вызовов функций `persist` и `checkpoint`, фреймворк Spark все равно во время перетасовки записывает на диск определенные данные. Файлы этих данных называются перетасовочными и обычно содержат все записи из каждой входной секции, отсортированные подпрограммой отображения. Обычно перетасовочные файлы остаются в локальных каталогах рабочих узлов на все время жизни приложения. Следовательно, при повторном использовании драйверной программой набора RDD, уже подвергнутого перетасовке, фреймворк Spark получает возможность избежать повторного вычисления этого набора вплоть до состояния на момент перетасовки за счет применения перетасовочных файлов.

В отличие от других типов кэширования мы не можем выяснить наличие перетасовочных файлов для конкретного набора RDD, например, не существует эквивалента команды `isCheckPointed`, которая возвращает `true`, если для данного набора RDD была создана контрольная точка. В целом, однако, перетасовочные файлы не очищаются явным образом до момента выхода RDD из области видимости. Тем не менее в определении того, не были ли подобным образом пропущены какие-либо этапы, может помочь пользовательский веб-интерфейс, как показано на рис. 5.3.

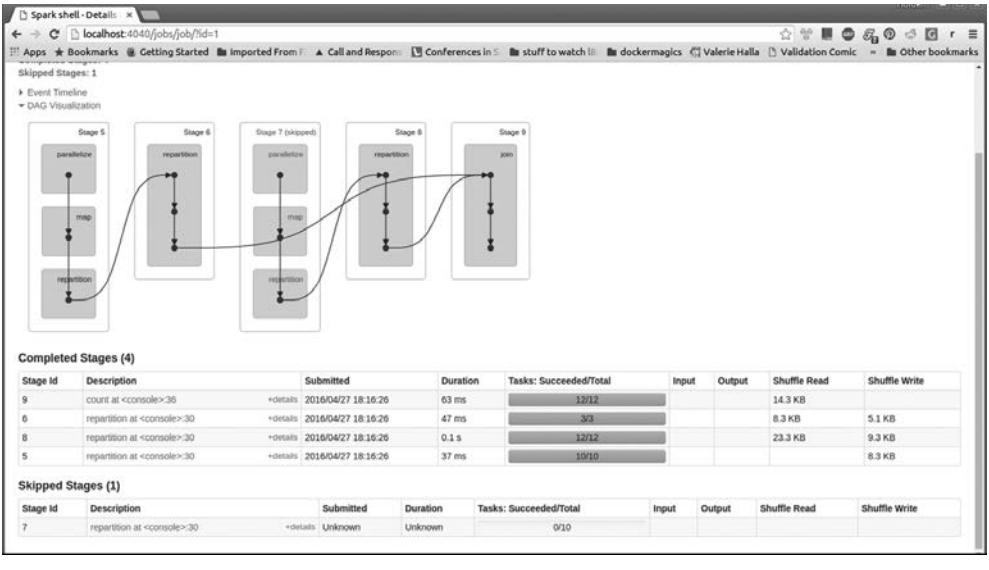

Рис. 5.3. Этап, пропущенный при чтении перетасовочных файлов

Производительность при повторном использовании перетасовочных файлов аналогична производительности кэшируемого только на уровне диска набора RDD.

Размер перетасовочных файлов может быть очень велик, а во фреймворке Spark отсутствует для них явное управление кэшем. А если оставлять ссылки на наборы RDD для перетасованных выходных данных, то можно получить ошибки нехватки места на диске, когда на драйвере не выполняется сборка мусора.

Ошибки нехватки места на диске возникают неожиданно, но в кластерах с небольшими объемами дискового пространства случаются удивительно часто. Их может вызывать длительно работающая среда командной оболочки, в которой не производится сборка мусора наборов RDD, созданных в самой верхней области видимости. Фреймворк записывает вывод операций перетасовки в файлы на дисках исполнителей в локальном каталоге Spark. Эти файлы удаляются, только когда

набор RDD подвергается сборке мусора, а это при большом объеме выделенной драйверной программе оперативной памяти происходит нечасто. Одно из решений — вручную инициировать сборку мусора (при условии, что наборы RDD вышли за пределы области видимости). Если DAG оказывается слишком длинным, то создание контрольных точек может помочь в обеспечении доступности наборов для сборки мусора.

Соображения по поводу зашумленных кластеров

Зашумленные кластеры и кластеры с большими объемами непрогнозируемого трафика — фундаментальная проблема Spark. По умолчанию он не сохраняет большую часть промежуточных результатов (за исключением этапа перетасовки). Следовательно, в случае вытеснения данных фреймворку придется повторить вычисления текущего задания вплоть до момента сбоя. В зашумленном кластере с постоянными прерываниями длительных заданий это представляет собой немалую проблему. Помочь хоть какому-то выполнению заданий может создание контрольных точек. Оно разбивает граф происхождения набора RDD, а значит, снижает стоимость повторного вычисления идущих далее преобразований. Кроме того, оно приводит к сохранению во внешнее хранилище, поэтому неожиданные отказы не влекут потери данных. Если сбои происходят регулярно, но не являются фатальными, то имеет смысл настроить задание так, чтобы выполнять сохранение на несколько машин с помощью, например, опции хранения `MEMORY_AND_DISK_2`, реплицирующей данные по двум машинам. Следовательно, не нужно будет выполнять повторное вычисление при сбоях в одном из узлов. Это особенно важно в случае очень дорогостоящих широких преобразований.

По умолчанию в Spark применяется парадигма FIFO (first in, first out — «первым вошел, первым вышел») для определения очередности заданий в системе. Это значит, что первое поступившее задание станет выполняться полностью, с приоритетным использованием всех имеющихся ресурсов. Однако если для выполнения задания не требуется весь кластер, то может быть запущено следующее. Планирование заданий на основе FIFO позволяет гарантировать, что требовательные к доступному пространству задания получат все необходимые им ресурсы. Но если запустить задание за несколько секунд до многочасового процесса, то стратегия FIFO вас сильно разочарует. Фреймворк Spark предоставляет недискриминационный планировщик, созданный по образу и подобию недискриминационного планировщика Hadoop, позволяющий более равномерно распределять ресурсы в кластерах с высокими объемами трафика. Недискриминационный планировщик распределяет задачи из различных заданий по исполнителям циклическим образом (то есть выделяя по нескольку задач исполнителям из каждого задания). При использовании недискриминационного планировщика короткое, маленькое задание может быть запущено до завершения ранее запущенного длительного.

Недискриминационный планировщик также поддерживает объединение заданий в пулы и назначение последним различных приоритетов (весов). Заданиям в пуле

выделяется одинаковое количество ресурсов, а самим пулам ресурсы выделяются в соответствии с их весами. Пулы — отличный способ гарантировать завершение наиболее важных или очень дорогостоящих заданий. Кроме того, этот планировщик обеспечивает равномерное распределение ресурсов пользователям вне зависимости от количества запущенных ими заданий. Прочитать более подробно о применении и настройке недискриминационного планировщика можно в документации по планированию заданий Spark (https://spark.apache.org/docs/latest/job-scheduling.html).

Взаимодействие с накопителями

Взаимное влияние кэширования и накопителей может существенно усложнить рассуждения по поводу накопителей. Как мы уже упоминали, при необходимости повторно вычислять часть набора RDD фреймворк Spark способен продолжать наращивать накопитель, приводя к двойному учету значений из вычисляемой повторно части. Более того, не все вычисления охватывают секцию целиком. Что удивительно, кэширование не предотвращает ни двойного учета, ни проблем, возникающих из-за частично вычисляемых секций. Вследствие вероятного вытеснения закэшированных секций двойной учет может произойти при сбое машины с закэшированными данными или при вытеснении секции (с целью освободить место для секции, закэшированной позднее). К сожалению, кэширование при использовании накопителей способно привести к тому, что задание, вроде бы вычислявшее правильное значение на небольшом фрагменте данных, вычисляет потом неправильное значение на большом объеме.

Резюме

Теперь, когда мы разобрались, как добиться максимальной отдачи от стандартных преобразований наборов RDD и соединений, пришло время изучить вопросы, связанные с самым важным и сложным подмножеством преобразований наборов — операциями с парами «ключ — значение». Не все методы, которые вы изучите, следует использовать во всех программах Spark, и ряд приведенных в этой главе готовых решений уместен скорее в случаях, когда определенные инструменты подходят плохо (см. подраздел «Накопители» на с. 130). При работе с данными типа «ключ — значение» применимы многие из методов и соображений для стандартных преобразований наборов RDD: если преобразование не зависит от ключа, то методы из данной главы могут оказаться еще более уместными.

6 Работа с данными типа «ключ — значение»

Как и любой хороший инструмент распределенных вычислений, фреймворк Spark активно задействует парадигму пар «ключ — значение» для определения и распараллеливания операций, особенно широких преобразований, требующих перераспределения операций по машинам. Возможности работы с данными типа «ключ — значение», позволяющие фреймворку легко распараллелить работу, очень полезны как для параллельного выполнения операций группировки, так и для распределения записей по машинам — идет ли речь о вычислении сводных показателей или слиянии пользовательских записей. В Spark имеется свой класс `PairRDDFunctions` с операциями для наборов RDD и кортежей. Доступный благодаря неявному преобразованию типов, этот класс содержит большую часть методов для соединений и пользовательских сводных показателей. Класс `OrderedRDDFunctions` включает методы для сортировки. Его можно применять для наборов RDD кортежей, в которых для первого элемента (ключа) задана неявная упорядоченность.

Аналогичные операции можно выполнять над наборами Dataset, как обсуждалось в подразделе «Группирующие операции с объектами Dataset» на с. 86.

Несмотря на удобство, операции с данными типа «ключ — значение» способны породить множество проблем с производительностью. На поверку наиболее дорогостоящие операции фреймворка Spark отлично вписываются в парадигму пар «ключ — значение». Так происходит потому, что самые широкие преобразования — это преобразования данных типа «ключ — значение», для хорошей производительности большинства из которых требуются достаточно тонкая настройка и аккуратность. Именно подобным вопросам производительности будет посвящена данная

глава. Мы надеемся не только предоставить вам справочник по использованию функций классов `PairRDDFunctions` и `OrderedRDDFunctions`, но и на основе выводов из глав 2 и 5 создать досконально руководство по вычислению фреймворком Spark широких преобразований, а также перепроектированию логики программы в целях достижения максимальной производительности задач, требующих упорядоченности данных.

В частности, операции над данными типа «ключ — значение» могут вызывать такие проблемы:

- ошибки нехватки памяти в драйвере;
- ошибки нехватки памяти на узлах исполнителей;
- сбои при перетасовке;
- «отставание задач» или особенно медленное вычисление отдельных секций.

Первая проблема обычно вызывается действиями. Мы обсудим вопросы производительности, связанные с действиями над данными типа «ключ — значение», в подразделе «Действия над парами “ключ — значение”» на с. 156. Остальные три проблемы с производительностью чаще всего возникают из-за перетасовок при выполнении широких преобразований в классах `PairRDDFunctions` и `OrderedRDDFunctions`. В этой главе мы сосредоточимся на двух основных методиках решения связанных с перетасовками проблем производительности, которые назовем «перетасовывать реже» и «перетасовывать лучше».

- *Перетасовывать реже.* Мы опишем методики минимизации количества перетасовок, необходимого для сложного вычисления. Один из способов минимизировать количество перетасовок в требующем нескольких преобразований вычислении — сохранять секционирование при узких преобразованиях (чтобы избежать перетасовки данных; см. подраздел «Сохранение информации о секционировании от преобразования к преобразованию» на с. 170). В некоторых случаях можно использовать один и тот же объект `Partitioner` для серии широких преобразований. Это особенно полезно в ситуациях, требующих избежать перетасовок во время соединений и снизить количество перетасовок, необходимых для вычисления серии широких преобразований (см. подразделы «Совместная группировка» на с. 165 и «Использование совместно расположенных и секционированных наборов RDD» на с. 171). Мы также обсудим применение пользовательских объектов `Partitioner` (см. подраздел «Пользовательское секционирование» на с. 169) в целях наиболее эффективного распределения данных для последующих вычислений, равно как и спуска работы по вычислениям на этап перетасовки для повышения производительности сложных вычислений (см. раздел «Вторичные сортировки и функция repartitionAndSortWithinPartitions» на с. 176).

- *Перетасовывать лучше*. Иногда выполнить вычисления без перетасовки невозможно. Однако не все широкие преобразования и перетасовки требуют одинаковых вычислительных затрат и одинаково подвержены ошибкам. Можно предотвратить ошибки нехватки памяти в исполнителях и ускорить выполнение широких преобразований, в особенности при агрегировании. Для этого нужно использовать такие широкие преобразования, как `reduceByKey` и `aggregateByKey`, не требующие загрузки всех записей для ключа в оперативную память и способные выполнить свертку при отображении (см. раздел «Чем так опасна функция groupByKey» на с. 157 и подраздел «Предотвращение ошибок нехватки памяти при операциях агрегирования» на с. 164). Наконец, перетасовка данных с равномерным распределением записей по ключам, причем содержащих большое количество различных ключей, предотвращает ошибки нехватки памяти в исполнителях и «отставание задач» (см. раздел «Выявление отстающих задач и несбалансированных данных» на с. 191).

Пример со Златовлаской

В этой главе мы будем неоднократно обращаться к проекту, над которым работали авторы книги, требующему вычисления произвольных ранговых статистик в многомерных и крупномасштабных данных, — он послужит образцом сложных преобразований данных типа «ключ — значение».

У клиента, которого мы назовем Златовлаской, имеются данные, отражающие тысячи различных показателей для сотен миллионов панд. Ее данные выглядят приблизительно так, как показано в табл. 6.1.

Таблица 6.1. Пример данных Златовласки

Имя панды	Счастье	Приятность	Доброта	Ласковость
Панда-мама	15,0	0,25	2467,0	0,0
Панда-папа	2,0	1000	35,4	0,0
Панда-сын	10,0	2,0	50,0	0,0
Игрушечная панда панды-сына	3,0	8,5	0,2	98,0

Атрибуты всех панд представлены в виде чисел типа `double`.

Златовласка хочет, чтобы мы разработали приложение, в которое можно было бы ввести произвольный список целых чисел `n1...nk` и получить в качестве результата `n`-й по рангу элемент каждого из столбцов. Например, если Златовласка вводит `8`, `1000` и `20 000 000`, то наша функция должна вернуть 8, 1000 и 20 000 000-ю по рангу панду для каждого из столбцов атрибутов.

Для иллюстрации этого образца допустим, что Златовласка хочет найти 2-й и 4-й элементы из табл. 6.1. Наша функция должна возвращать нечто напоминающее табл. 6.2.

Таблица 6.2. Пример результата Златовласки

Название столбца	Индекс столбца	Ранговые статистики
Счастье	1	List(3.0, 15.0)
Приятность	2	List(2.0, 1000.0)
Доброта	3	List(35.4, 2467.0)
Ласковость	4	List(0.0, 98.0)

Мы назвали нашего клиента Златовлаской потому, что она очень требовательна и ее дом (то есть ее кластер) битком набит медведями[1] (другими клиентами). В данном случае Златовласку едва ли устроят приблизительные границы квантилей, она потребует, чтобы результаты нашей функции были значениями из первоначального набора данных. Следовательно, задача по своей сути вычислительно сложна, так как требует сортировки определенным образом всех значений в каждом из столбцов.

Поскольку данные носят столбцовый характер, то можно попытаться решить задачу с помощью Spark SQL. Однако в его ранних версиях не было никакой поддержки ранговых статистик. Для решения задачи можно попробовать написать UDF/UDAF, но такой вариант окажется весьма громоздким, поскольку сценарий применения непрост и построчные вычисления невозможны. Следовательно, решение должно использовать Spark Core[2].

Златовласка. Версия 0: итеративное решение

Первое приходящее в голову решение этой задачи: цикл по всем столбцам с отображением каждой строки в отдельное значение, применением к каждому из столбцов функций `sortBy` и `zipWithIndex` фреймворка Spark и фильтрации индексов, соответствующих требуемым ранговым статистикам.

---

[1] Речь идет о героине английской сказки Goldilocks and the three bears («Златовласка и три медведя»). — *Примеч. ред.*

[2] Как мы надеемся продемонстрировать в этой главе, даже в случаях, когда готовые для использования решения из пакета Spark SQL подходят для конкретного сценария, предоставляемые Spark Core возможности контроля зачастую позволяют создать для него лучшее решение. Итоговое решение, описанное в данной главе, безусловно, способно выполнять весьма изящные свертки, что делает его более быстрым, чем даже очень хорошо реализованные ранговые статистики из Spark SQL.

Для простоты предположим: столбцовые данные читаются из устойчивого хранилища, скажем набора DataFrame; строки все сформированы корректно и строковый столбец с именами панд был удален. Соответственно, наша функция принимает на входе набор DataFrame всех столбцов чисел типа double (отражающих данные о пандах) и список длинных целых чисел, отражающих местоположение искомых элементов в каждом столбце, допустим 1-й, 1000-й. Функция должна возвращать ассоциативный массив соответствий индексов столбцов списку ранговых статистик из этого столбца.

Пример 6.1 представляет собой реализацию первого решения задачи Златовласки, в котором мы проходим в цикле по всем столбцам и сортируем каждый из них с помощью распределенной сортировки фреймворка Spark.

Пример 6.1. Златовласка. Версия 0: итеративное решение

```
def findRankStatistics(
  dataFrame: DataFrame,
  ranks: List[Long]): Map[Int, Iterable[Double]] = {
  require(ranks.forall(_ > 0))
  val numberOfColumns = dataFrame.schema.length
  var i = 0
  var  result = Map[Int, Iterable[Double]]()

  while(i < numberOfColumns){
    val col = dataFrame.rdd.map(row => row.getDouble(i))
    val sortedCol : RDD[(Double, Long)] = col.sortBy(v => v).zipWithIndex()
    val ranksOnly = sortedCol.filter{
      // Ранговые статистики проиндексированы с 1;
      // например, первый элемент – 0
      case (colValue, index) =>  ranks.contains(index + 1)
    }.keys
    val list = ranksOnly.collect()
    result += (i -> list)
     i+=1
  }
  result
}
```

Это решение работает и является относительно отказоустойчивым, но очень медленным, поскольку необходима итеративная сортировка данных по каждому столбцу. Другими словами, при 8000 столбцов понадобилось бы 8000 сортировок.

Каковы же лучшие варианты решения?

Поскольку для каждой из сортировок не нужна информация о других сортировках, интуитивно понятно, что можно распараллелить это вычисление, взяв за единицу параллелизма отдельный столбец. Можно представить данные в виде одного длинного списка пар «ключ — значение», в которых ключи представляют индексы столбцов. После этого вычисление производится параллельно по ключам.

Отобразим данные из табл. 6.1 в виде следующего списка пар «ключ — значение»:

```
(ключ, значение)
(1, 15.0)
(2, 0.25)
(3, 2467.0)
(4, 0.0)
(1, 2.0)
(2, 1000.0)
(3, 35.4)
(4, 0.0)
(1, 10.0)
(2, 2.0)
(3, 50.0)
(4, 0.0)
(1, 3.0)
(2, 8.5)
(3, 0.2)
(4, 98.0)
```

Если прочитать наши данные в виде набора `DataFrame`, то можно выполнить это отображение с помощью простой функции, показанной в примере 6.2.

Пример 6.2. Златовласка. Версия 1: отображение по парам (индекс столбца, значение)

```
def mapToKeyValuePairs(dataFrame: DataFrame): RDD[(Int, Double)] = {
  val rowLength = dataFrame.schema.length
  dataFrame.rdd.flatMap(
    row => Range(0, rowLength).map(i => (i, row.getDouble(i)))
  )
}
```

Операция flatMap фреймворка Spark имитирует поведение операции flatMap, определенной для итераторов и коллекций языка Scala. Операция flatMap — исключительно универсальное узкое преобразование, способное, однако, вызвать затруднения у тех, кто плохо знаком с языком Scala. Она позволяет задать отображение всех записей в коллекции элементов с последующим объединением получившихся коллекций. В данном случае задается отображение объекта Row Spark SQL в последовательность элементов, а именно пар (индекс столбца, значение). В итоговом наборе RDD будет больше записей, чем в предыдущем; каждая запись будет состоять из пар (индекс столбца, значение). Операция flatMap, вообще говоря, не требует увеличения общего количества записей. На самом деле она может быть особенно полезна именно потому, что, в отличие от операции map, позволяет возвращать пустые коллекции для некоторых элементов. Следовательно, эта операция применима для фильтрации и преобразования элементов за один проход. Другими словами, выполнение операций map и filter над одним набором RDD всегда можно объединить в один шаг flatMap.

После использования данной функции можно распараллелить вычисление по индексам столбцов (в данном случае индексы столбцов — ключи записей). Задача Златовласки, если ее сформулировать подобным образом, становится задачей пар «ключ — значение», а именно: *разработать функцию, которая принимала бы на входе набор RDD пар целых чисел/чисел типа double и список длинных целых, n1... nk и возвращала ассоциативный массив, содержащий отображение ключей в список k чисел типа double, представляющих собой n1-й, n2-й... nk-й элементы для этого индекса столбца (ключа)*.

Как использовать классы PairRDDFunctions и OrderedRDDFunctions

Если вы уже давно работаете с фреймворком Spark, то наверняка знакомы с классами `PairRDDFunctions` и `OrderedRDDFunctions`. Тем не менее мы вкратце расскажем, как их задействовать. Если же эти функции вам не знакомы, то глава 4 «Работа с парами "ключ — значение"» нашей книги *Learning Spark* содержит очень хорошее их описание. Класс RDD фреймворка Spark применяет неявные функции преобразований типов языка Scala, так что функции класса `PairRDDFunctions` будут доступны для любого набора RDD типа `(K,V)`[1]. В случае класса `PairRDDFunctions` типы `K` и `V` могут быть любыми, но для класса `OrderedRDDFunctions` (функции `sortByKey`, `repartitionAndSortWithinPartitions`, `filterByRange`) у `K` должна быть какая-то неявная упорядоченность. Для большинства распространенных типов, например числовых типов или строк, в языке Scala упорядоченность уже определена. В случае же работы с пользовательским типом может потребоваться задать ее самостоятельно. Фреймворк Spark применяет неявное преобразование, чтобы преобразовать набор RDD, соответствующий требованиям классов `PairRDDFunctions` и `OrderedRDDFunctions`, из обобщенного типа в тип `PairRDD` или `OrderedRDD`. Для этого неявного преобразования типа следует импортировать соответствующую библиотеку. Так, для использования класса `PairRDDFunctions` фреймворка Spark необходимо импортировать `SparkContext`; то есть список импортов должен включать `import org.apache.spark.SparkContext._`.

При написании функции, использующей класс OrderedRDDFunctions с ключами обобщенного типа, может понадобиться включить в нее код определения неявной переменной типа ordering. В образце со вторичной сортировкой, который мы обсудим в разделе «Вторичные сортировки и функция repartitionAndSortWithinPartitions» на с. 176, мы зададим упорядоченность для объекта pandaKey так, как показано в примере 6.3.

[1] Пары (ключ, значение). — *Примеч. пер.*

Пример 6.3. Определение неявной упорядоченности для работы с классом OrderedRDDFunctions

```
implicit def orderByLocationAndName[A <: PandaKey]: Ordering[A] = {
  Ordering.by(pandaKey => (pandaKey.city, pandaKey.zip, pandaKey.name))
}

implicit val ordering: Ordering[(K, S)] = Ordering.Tuple2
```

Действия над парами «ключ — значение»

В подразделе «Функции, применяемые к наборам RDD: преобразования и действия» на с. 35 мы обсуждали, как вычисляются преобразования в исполнителях Spark при вызове действия. Мы также объяснили, что действия обычно приводят к перемещению данных из исполнителей Spark с помощью или отправки их на драйвер, или записи в устойчивое хранилище. Вообще говоря, рекомендуется с большой осторожностью относиться к действиям, возвращающим неограниченные объемы данных на драйвер, поскольку они могут приводить там к ошибкам нехватки памяти. Большинство действий над парами «ключ — значение» (включая `countByKey`, `countByValue`, `lookUp` и `collectAsMap`) возвращают данные на драйвер. Причем в большинстве случаев объем возвращаемых ими данных потенциально неограничен, поскольку количество ключей и значений неизвестны. Например, действие `countByKey` возвращает по точке данных для каждого ключа, а это способно привести к ошибкам нехватки памяти, если количество различных ключей больше, чем может поместиться в оперативной памяти драйвера. Напротив, `lookUp` возвращает все значения для каждого из ключей, поэтому способно повлечь ошибки нехватки памяти, если объем данных в одном из ключей таков, что не помещается в оперативной памяти драйвера.

Операция lookUp является дорогостоящей еще и потому, что запускает перетасовку в случае, если неизвестен объект Partitioner набора RDD.

Помимо количества записей, важным фактором, влияющим на частоту ошибок нехватки памяти, является размер каждой из них. Например, если каждая из записей представляет собой пользовательский объект или тип-коллекцию, то действие, которое успешно выполнило сбор аналогичного количества записей в набор RDD байтов, может завершиться сбоем. В целом желательно проектировать задачи над данными типа «ключ — значение» таким образом, чтобы ключи помещались в оперативной памяти драйвера. Значения должны быть по крайней мере хорошо распределены по ключам, а лучше — распределены так, чтобы записи для любого ключа помещались в оперативной памяти любого исполнителя. Мы обсудим последствия плохого распределения ключей в разделе «Выявление отстающих задач

и несбалансированных данных» на с. 191 и дадим несколько советов о действиях в случае асимметричных данных. Как и для всех программ Spark, нужно стараться выполнить преобразования для уменьшения объема данных, прежде чем вызывать действия, перемещающие результаты на драйвер.

Преобразования данных типа «ключ — значение» могут приводить к ошибкам нехватки памяти обычно в исполнителях, если требуют хранения всех относящихся к одному ключу данных в оперативной памяти одной секции. Избежать ошибок нехватки памяти и уменьшить количество перетасовок путем оптимизации преобразований несколько сложнее, чем избежать проблем с действиями. Поэтому мы сосредоточимся именно на преобразованиях данных типа «ключ — значение».

Чем так опасна функция groupByKey

Многие источники — включая документацию фреймворка Spark — предостерегают, что функция `groupByKey`, возвращающая итератор каждого элемента по ключу, масштабируется очень плохо. В этом разделе мы попытаемся пояснить, в каких случаях данная функция вызывает проблемы при масштабировании. Мы также хотели бы посоветовать некоторые альтернативы функции `groupByKey`. Для начала предстоит пересмотреть сценарий примера со Златовлаской, поскольку наше первое решение этой задачи будет применять `groupByKey`.

Златовласка. Версия 1: решение с использованием функции groupByKey

Одно из простейших решений задачи Златовласки — задействовать функцию `groupByKey` для группировки элементов каждого столбца. Функция `groupByKey` возвращает итератор элементов по ключу, так что для сортировки элементов по ключу необходимо преобразовать итератор в массив, после чего отсортировать последний[1]. Затем можно отфильтровать этот массив, получив элементы, соответствующие нашим ранговым статистикам.

Пример 6.4 — реализация решения с использованием функции `groupByKey`. Ради единообразия эта функция также принимает на входе набор `DataFrame` и список

[1] Возможен лишь однократный проход итератора. Следовательно, итераторы нельзя использовать без преобразования их типа, поскольку сортировка требует нескольких проходов по данным. Их необходимо сначала преобразовать в другую структуру данных. Подробнее об итераторах, их преимуществах и ограничениях можно прочитать в разделе «Выполнение преобразований “итератор — итератор” с помощью функции mapPartitions» на с. 121.

позиций элементов в виде значений типа long. Она вызывает функцию, создающую пары «ключ — значение», описанные в примере 6.2.

Пример 6.4. Златовласка. Версия 1, решение с использованием функции groupByKey

```
def findRankStatistics(
  dataFrame: DataFrame,
  ranks: List[Long]): Map[Int, Iterable[Double]] = {
  require(ranks.forall(_ > 0))
  // Отображение в пары (индекс столбца, значение)
  val pairRDD: RDD[(Int, Double)] = mapToKeyValuePairs(dataFrame)

  val groupColumns: RDD[(Int, Iterable[Double])] = pairRDD.groupByKey()
  groupColumns.mapValues(
    iter => {
      // Преобразование в массив и сортировка
      val sortedIter = iter.toArray.sorted

      sortedIter.toIterable.zipWithIndex.flatMap({
      case (colValue, index) =>
          if (ranks.contains(index + 1)) {
            Iterator(colValue)
          } else {
            Iterator.empty
          }
    })
  }).collectAsMap()
}

def findRankStatistics(
  pairRDD: RDD[(Int, Double)],
  ranks: List[Long]): Map[Int, Iterable[Double]] = {
  assert(ranks.forall(_ > 0))
  pairRDD.groupByKey().mapValues(iter => {
    val sortedIter  = iter.toArray.sorted
    sortedIter.zipWithIndex.flatMap(
    {
      case (colValue, index) =>
        if (ranks.contains(index + 1)) {
          // Одна из требуемых ранговых статистик
          Iterator(colValue)
          } else {
            Iterator.empty
          }
      }
    ).toIterable // Преобразование в более обобщенный итерируемый тип,
                 // лучше подходящий под наши требования
  }).collectAsMap()
}
```

У этого решения есть несколько преимуществ. Во-первых, оно возвращает правильный ответ. Во-вторых, оно короткое и понятное. В нем применяются готовые

функции фреймворка Spark и языка Scala, для него существует очень немного граничных случаев, и его относительно легко тестировать. При небольшом объеме данных, особенно если во входных данных много столбцов, но мало записей, решение работает весьма эффективно, поскольку требует лишь одной перетасовки на шаге `groupByKey` и шаг сортировки можно вычислить в исполнителях как узкое преобразование.

В этой функции используется функция collectAsMap, которой присущи те же проблемы, что и функции collect, упомянутой ранее. Однако в данном случае опасность возникновения ошибок нехватки памяти минимальна, поскольку на момент сбора известно количество как ключей, так и значений для каждого из них. Количество ключей в точности равно количеству столбцов, которое не должно превышать несколько тысяч. А количество значений равно длине списка ранговых статистик на входе функции, изначально хранящегося не в распределенном виде. Однако хорошей практикой будет задать ограничение размера входного списка, чтобы предотвратить сбои на шаге сбора.

В используемой среде и при объеме данных 10 000 строк и несколько тысяч столбцов это решение работает на порядок быстрее, чем представленное в разделе «Пример со Златовлаской» на с. 151, в котором мы итеративно проходили в цикле по столбцам, сортируя каждый из них. Однако при объемах данных в несколько миллионов строк оказывается, что данное решение также приводит к ошибкам нехватки памяти даже в многоузловом кластере.

Почему операция groupByKey вызывает ошибки

Если вы читали *Learning Spark* или провели немало времени, работая с масштабными приложениями на Spark, то результаты использования функции `groupByKey` для решения задачи Златовласки вас вряд ли удивят, ведь упомянутая функция печально известна ошибками нехватки памяти в случае больших объемов данных. Причина заключается в том, что создаваемые функцией `groupByKey` «группы» всегда представляют собой итераторы, которые нельзя сделать распределенными. Это приводит к дорогостоящему шагу «перетасовочного чтения», когда фреймворку Spark приходится читать все перетасовочные данные с диска в память.

На рис. 6.1 приведен снимок экрана веб-интерфейса Spark, иллюстрирующий высокий уровень затрат функции `groupByKey`.

Обратите внимание: объем читаемых при перетасовке данных в процессе этого вычисления составляет 86 Мбайт, хотя объем входных данных — около 200.

Другими словами, Spark приходится считывать почти все перетасовочные данные в оперативную память.

Вследствие секционирования по хеш-значению и «подтягивания» результатов в оперативную память для группировки в виде итераторов функция `groupByKey` часто приводит к ошибкам нехватки памяти исполнителей, когда количество дублирующихся записей на один ключ велико. Все записи, хеш-значения ключей которых совпадают, должны находиться в оперативной памяти одной машины. Следовательно, если хотя бы один из ключей содержит так много записей, что они не помещаются в памяти одного исполнителя, то вся операция завершится неудачей.

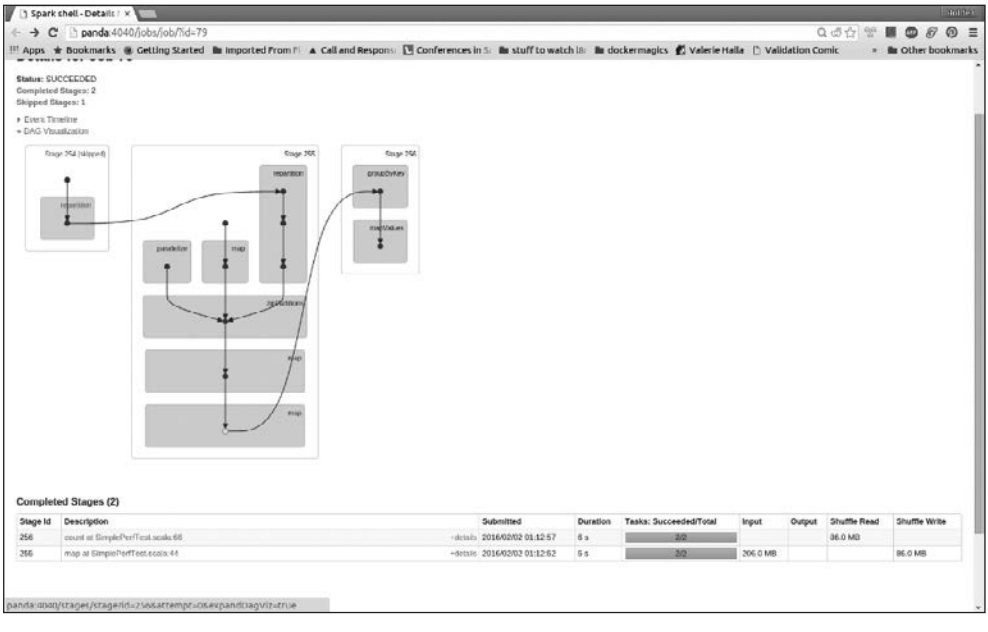

Рис. 6.1. DAG функции groupByKey и перетасовочное чтение

Рисунок 6.2 иллюстрирует операцию `groupByKey` над данными о закрученных вверх усах. Как вы можете видеть, больше всего записей, соответствующих почтовому индексу 94110, представляет собой почтовый индекс округа Мишн в Сан-Франциско. Хотя при равномерном распределении записи поместились бы в исполнителях, все относящиеся к почтовому индексу 94110 записи не поместятся в одном исполнителе после шага `groupByKey`.

Вообще говоря, лучше выбирать агрегирующие операции, которые могли бы перед перетасовкой выполнять объединение при отображении с целью уменьшить количество записей из расчета на ключ (например, `aggregateByKey` или `reduceByKey`).

Если это невозможно, то хорошей альтернативой `groupByKey` будет использование широкого преобразования, не требующего хранения в оперативной памяти всех относящихся к одному ключу значений, как будет обсуждаться в разделе «Вторичные сортировки и функция repartitionAndSortWithinPartitions» на с. 176. При необходимости использовать `groupByKey` лучше сделать следующей за ним операцией преобразование «итератор — итератор», как обсуждалось в разделе «Выполнение преобразований "итератор — итератор" с помощью функции mapPartitions» на с. 121.

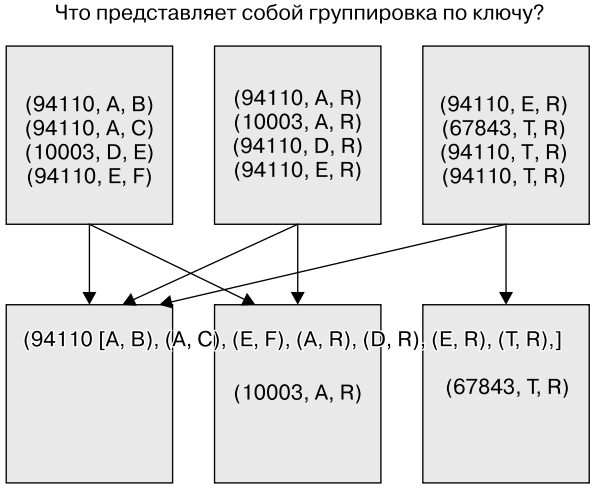

Рис. 6.2. «Перебор» памяти при операции groupByKey

Выбор операции агрегирования

Перетасовка записей с целью объединить записи с одинаковыми ключами — распространенный сценарий использования операций Spark с данными типа «ключ — значение», а фреймворк предоставляет достаточно много подобных операций. Большинство из них являются надстройками над обобщенной операцией `combineByKey`, но сильно различаются по производительности. В данном разделе мы подробно рассмотрим эти операции, а также некоторые связанные с ними вопросы производительности.

Список операций агрегирования с комментариями по производительности

Операции агрегирования имеют конкретные показатели производительности, которые мы кратко опишем в табл. 6.3.

Таблица 6.3. Перечень операций агрегирования над парами «ключ — значение» фреймворка Spark

Функция	Назначение	Ограничения на ключи	Когда выходит за пределы памяти	Когда начинает работать медленно	Выходной объект Partitioner
groupByKey	Группировка значений с одинаковым ключом в единый итератор	Использование ключей-массивов при HashPartitioner по умолчанию. Для этого необходим пользовательский объект Partitioner	Когда все относящиеся к какому-либо отдельному ключу значения занимают слишком много места в оперативной памяти, чтобы их можно было прочитать с диска на одном исполнителе	Если объект Partitioner неизвестен, то это приводит к перетасовке. Ее стоимость растет по мере увеличения количества различных ключей, записей из расчета на ключ или в случае неравномерного распределения записей по ключам	По умолчанию HashPartitioner, но поддерживает и пользовательское секционирование
combineByKey	Объединение значений с одинаковым ключом с возвращением результата другого типа	Такие же, как и выше	Когда процедура «объединения по» использует слишком много памяти или генерирует слишком много накладных расходов по сборке мусора, либо размер накопителя для одного из ключей становится слишком большим (это та же проблема, что и в операции groupByKey)	Аналогично вышеуказанному, но может работать быстрее, чем операция groupByKey, если операция объединения представляет собой свертку	См. выше
aggregateByKey	Аналогично combineByKey, но для всех накопителей используется одно нулевое значение	Такие же, как и выше	Как и выше, но при хорошей реализации вероятность возникновения ошибок сборки мусора меньше (см. раздел «Минимизация количества создаваемых объектов» на с. 114	См. выше, но обычно работает быстрее, чем операция combineByKey, благодаря выполнению слияния при отображении перед отправкой данных на объединение	См. выше

Функция	Назначение	Ограничения на ключи	Когда выходит за пределы памяти	Когда начинает работать медленно	Выходной объект Partitioner
reduceByKey	Объединение значений с одинаковым ключом. Свертка должна производиться к тому же типу, что и у исходных значений	См. выше, но часто оказывается менее дорогостоящей, чем combineByKey, поскольку aggregateByKey поддерживает повторное использование объекта накопителя, чтобы избежать создания лишних объектов	См. выше, но ограничения по типу снижают вероятность ошибок нехватки памяти. Если объединение не является типом-коллекцией, то данная функция, вероятнее всего, будет выполнять свертку. Затраты на сборку мусора меньше, чем у функции aggregateByKey, поскольку не создается никаких дополнительных объектов-накопителей	Аналогично aggregateByKey	См. выше
foldByKey	Объединение значений с одинаковым ключом с помощью ассоциативной функции объединения и нулевого значения, которое можно прибавлять к результату произвольное количество раз. Используется вместо reduceByKey, если для типа данных существует естественный 0	См. выше	См. выше	См. выше. Производительность практически идентична операции reduceByKey	См. выше

Чтобы избежать выделения памяти в функции aggregateByKey, меняйте значение старого накопителя вместо возвращения нового. См. раздел «Минимизация количества создаваемых объектов» на с. 114.

Предотвращение ошибок нехватки памяти при операциях агрегирования

Функция `combineByKey` и все основанные на ней операторы агрегирования (`reduceByKey`, `foldLeft`, `foldRight`, `aggregateByKey`) не лучше функции `groupByKey` в смысле ошибок памяти, *если* приводят к излишнему увеличению размера накопителей для отдельных ключей. На самом деле при взгляде на реализацию функции `groupByKey` можно увидеть, что она реализована с помощью `combineByKey`, в которой накопитель представляет собой итератор со всеми данными. Следовательно, накопитель соответствует размеру всех данных для определенного ключа. Другими словами, вероятность возникновения ошибок нехватки памяти вследствие этих операций невелика в случае уменьшения размера данных на шагах объединения. Однако рост накопителя при добавлении новых записей в конце концов приведет к ошибкам нехватки памяти, если одному ключу соответствует много записей.

Представим приблизительный объем памяти для операций формирования последовательности и объединения.

При операции формирования последовательности:

```
SeqOp(acc, v) => acc'
```

и операции объединения:

```
combOp(acc1, acc2) = acc3.
```

Вычислим, выполняются ли неравенства:

```
memory(acc') < memory(acc) + memory(v)
and memory(acc3) < memory(acc1) + memory(acc2).
```

Если да, то функция, скорее всего, представляет собой свертку. Если нет, то, как в случае функции `groupByKey`, лучше обдумать другую стратегию.

Помимо более низкой вероятности нехватки памяти, чем у `groupByKey`, следующие четыре функции — `reduceByKey`, `treeAggregate`, `aggregateByKey` и `foldByKey` — реализованы в расчете на применение объединения при отображении, то есть объединения записей с одинаковыми ключами до перетасовки. Эти может весьма существенно снизить объем читаемых при перетасовке данных. Сравним объем читаемых при перетасовке данных в нашей исходной операции `groupByKey` (рис. 6.1) и операции `reduceByKey` на рис. 6.3. Напомним, что в случае `groupByKey` объем читаемых при перетасовке данных был близок к объему входных. Однако использование для тех же входных данных функции `reduceByKey` снижает их количество на несколько сотен килобайт!

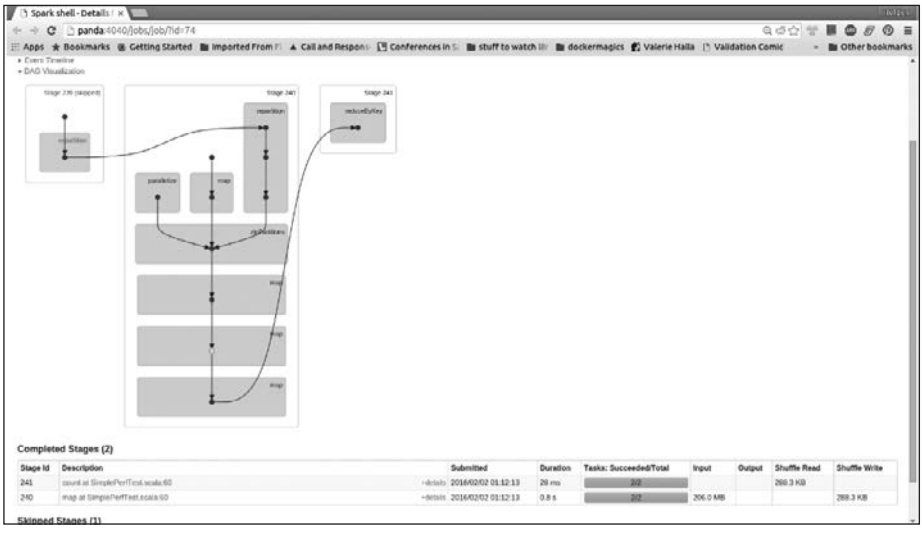

Рис. 6.3. DAG функции reduceByKey и чтение перетасовочных данных

Операции с несколькими наборами RDD

Некоторые преобразования могут выполняться с несколькими входными наборами RDD. Очевидно, что к ним относятся операции соединения, но они далеко не единственные.

Совместная группировка. Аналогично тому как все операции с накопителями (`reduceByKey`, `aggregateByKey`, `foldByKey`) реализованы с помощью функции `combineByKey`, так и все операции соединения реализованы благодаря функции `cogroup`, использующей тип `CoGroupedRDD`. Последний создается из последовательности наборов RDD пар «ключ — значение» с одним и тем же типом ключа. Функция `cogroup` перетасовывает все наборы RDD таким образом, чтобы элементы с одинаковым значением из всех наборов оказались в одной секции и в одном RDD в соответствии с ключом.

В классе `PairRDDFunctions` имеется несколько сигнатур функции `cogroup`. Последняя и ее функция-псевдоним `groupWith` могут принимать в качестве аргументов один, два или три набора RDD с одинаковым типом ключа (независимо от типа значений) и возвращать RDD со всеми ключами, а также кортежем объектов типа `Iterable`, где каждый объект `Iterable` представляет все значения из входных наборов RDD для этого ключа.

Допустим, у нас есть два набора данных с информацией обо всех пандах: один — с рейтингами в серии игр, а второй — с их любимыми блюдами. Можно задействовать функцию `cogroup`, чтобы связать идентификаторы панд с итератором их рейтингов и еще одним итератором их любимых блюд, как показано в примере 6.5.

Пример 6.5. Использование функции cogroup

```
val cogroupedRDD: RDD[(Long, (Iterable[Double], Iterable[String]))] =
  scoreRDD.cogroup(foodRDD)
```

Функция `cogroup` может быть полезна в качестве альтернативы соединению, при сочетании с несколькими наборами RDD. Вместо соединения нескольких наборов с одним эффективнее будет совместно секционировать наборы, поскольку это предотвратит перетасовку неоднократно соединяемых наборов.

Скажем, при необходимости соединить данные по рейтингам панд как с адресами, так и с любимыми блюдами лучше применить `cogroup`, чем выполнять две операции соединения, как показано в примере 6.6.

Пример 6.6. Использование функции cogroup с целью избежать выполнения нескольких соединений над одним набором

```
val addressScoreFood = addressRDD.cogroup(scoreRDD, foodRDD)
```

Несмотря на все ее преимущества, функция `cogroup` вызывает ошибки нехватки памяти по тем же причинам, что и `groupByKey`, если объем данных, относящихся хотя бы к одному из ключей в одном или обоих объединяемых наборах RDD, больше, чем помещается в отдельной секции. В частности, функция `cogroup` требует, чтобы все относящиеся к одному ключу записи всех группируемых совместно наборов RDD помещались в одной секции. Больше информации о соединениях приведено в главе 4.

Методы секционирования и данные типа «ключ — значение»

Как мы описывали в главе 2, секция в Spark представляет собой единицу параллельного выполнения, соответствующую одной задаче. В наборе RDD, не имеющем информации о секционировании, данные будут распределяться по секциям в соответствии лишь с размерами данных и секций[1]. Объект `Partitioner` определяет отображение записей набора RDD в индекс секции. Задав для RDD объект `Partitioner`, мы можем гарантировать, что записи каждой из секций, например, попадут в опре-

[1] Очень часто наборами RDD, у которых отсутствует объект Partitioner, оказываются загруженные из хранилища наборы. В таком случае данные набора чаще всего секционированы так же, как и само хранилище (например, Splits в Hadoop). Однако после чтения этих данных в Spark оказывается, что фреймворк ничего не знает об исходном секционировании и, следовательно, не может воспользоваться данной информацией. Этим, в частности, отличается создание контрольных точек от простого сохранения набора RDD в устойчивое хранилище с последующим чтением его оттуда. При создании контрольной точки Spark сохраняет определенные метаданные относительно набора, включая, если это применимо в конкретном случае, объект Partitioner.

деленный диапазон (метод секционирования по диапазону) или содержат только элементы с одинаковым хеш-кодом (метод секционирования по хешу).

Существует три метода, предназначенных исключительно для изменения способа секционирования набора RDD. Для наборов с обобщенным типом записей изменить количество используемых RDD секций можно с помощью методов `repartition` и `coalesce`, независимо от значений записей в нем. Как мы обсуждали в подразделе «Особый случай операции coalesce» на с. 112, метод `repartition` перетасовывает набор RDD, секционируя его по хешу с заданным количеством секций. (Мы поясним подробнее, как работает хеш-секционирование, в одноименном подразделе на с. 168.) Метод `coalesce`, с другой стороны, представляет собой оптимизированную версию метода `repartition`, которая обходится без полной перетасовки в случае, если желаемое количество секций меньше текущего. Напомним, что функция `coalesce` уменьшает количество секций, просто их объединяя, — и, следовательно, она не является широким преобразованием, поскольку секции определяются во время проектирования. Поведение `coalesce` при увеличении количества секций аналогично поведению `repartition`. Для наборов RDD пар «ключ — значение» можно использовать функцию `partitionBy`, принимающую на входе объект `Partitioner` вместо количества секций и перетасовывающую набор, секционируя его заданным образом. Метод `partitionBy` обеспечивает гораздо больше возможностей контроля над секционированием записей, поскольку объект `Partitioner` поддерживает описание функции, которая задает секции для записей на основе значения ключа.

Во всех случаях методы `repartition` и `coalesce` не присваивают набору RDD известный объект `Partitioner`. А вот использование метода `partitionBy` (и большинства других функций над парами «ключ — значение», вызывающих перетасовку) приводит к набору RDD с известным объектом `Partitioner`. В некоторых случаях, когда `Partitioner` набора известен, Spark может задействовать содержащуюся в нем информацию о локальности данных, чтобы избежать перетасовки, даже при наличии у преобразования широких зависимостей. Остальные разделы этой главы будут посвящены тому, как применять информацию о секционировании или пользовательское секционирование с целью «перетасовывать реже» и «перетасовывать лучше», а именно: мы хотели бы использовать эту информацию, чтобы уменьшить количество случаев, когда программа инициирует перетасовку, расстояние, на которое при перетасовке перемещаются данные, а также снизить вероятность отправки в одну из секций непропорционально большого количества данных, что повлечет ошибки нехватки памяти или отстающие задачи.

Использование объекта Partitioner фреймворка Spark

По сути, объект `Partitioner` определяет, как будут распределены записи, а следовательно, какая задача выполнит обработку тех или иных записей. Фактически объект `Partitioner` представляет собой интерфейс с двумя методами: `numPartitions`

и `getPartition`. Первый устанавливает, сколько будет секций в наборе RDD после секционирования. Второй определяет отображение ключей в числовые индексы секций, в которые должны попасть записи с этими ключами. Фреймворк Spark предоставляет две реализации интерфейса `Partitioner`: `HashPartitioner` и `RangePartitioner`. Если ни одна из них вас не устраивает, то можете задать свой пользовательский объект `Partitioner`.

Хеш-секционирование

Объект `Partitioner` по умолчанию для операций с наборами RDD пар «ключ — значение» (не операций с упорядоченными наборами RDD) — объект `HashPartitioner`. Последний определяет индексы дочерних секций на основе хеш-значения ключа. Объекту `HashPartitioner` необходим параметр `partitions`, определяющий количество секций в выходном наборе RDD и количество используемых в хеш-функции интервалов значений. Если этот параметр не задать, то Spark определит количество секций на основе значения свойства `spark.default.parallelism` из объекта `SparkConf`. При отсутствии заданного значения данного свойства Spark использует по умолчанию максимальное количество секций из «родословной» этого набора RDD. При широких преобразованиях с хеш-секционированием, например `aggregateByKey`, применяется необязательный параметр: количество секций, в качестве аргумента при создании объекта типа `HashPartitioner`. Более подробную информацию и советы по установке значения свойства `spark.default.parallelism` и выбору количества секций вы можете найти в подразделе «Количество и размер секций» на с. 314.

Секционирование по диапазонам значений ключей

При таком секционировании записи с относящимися к одному диапазону ключами распределяются в одну секцию. Данное секционирование необходимо для сортировки, так как позволяет гарантировать сортировку всего набора RDD с помощью сортировки записей внутри секций. Объект `RangePartitioner` сначала определяет границы диапазонов для всех секций путем выборочного исследования: оптимизации с целью равномерно распределить записи по секциям. Далее все записи из набора RDD перемещаются в секции, границы диапазонов которых включают соответствующие ключи. Сильно несбалансированные данные (для одних ключей много значений, а для других — нет вообще, с неравномерным распределением ключей) снижают точность выборки — и, как мы уже говорили, неравномерное секционирование замедляет выполнение задач, лежащих далее по конвейеру, приводя к возникновению «отстающих задач». А если дублирующихся ключей для всех относящихся к одному ключу записей слишком много и они не помещаются

в одном исполнителе, то секционирование по диапазонам значений ключей, как и хеш-секционирование, может вызывать ошибки нехватки памяти. Именно с этим обычно связаны проблемы производительности при сортировке.

Для создания объекта `RangePartitioner` с помощью фреймворка Spark необходимо не только количество секций, но и сам набор RDD для осуществления выборки. Он должен быть набором RDD кортежей, а для ключей следует задать упорядочение.

Выборка фактически требует частичного вычисления набора RDD, приводя, таким образом, к разрыву графа выполнения. Следовательно, секционирование по диапазонам значений ключей на самом деле представляет собой и преобразование, и действие. Дополнительные затраты на выборку означают следующее: в целом секционирование по диапазонам значений ключей — более дорогостоящее действие, чем хеш-секционирование. Требование упорядоченности ключей подразумевает, что секционирование по диапазонам значений ключей возможно выполнить не на всех наборах RDD кортежей. Следовательно, операции над парами «ключ — значение» (например, агрегирующие функции), при которых записи с одинаковыми ключами должны находиться на одной машине, но могут быть не упорядочены, применяют по умолчанию объект `HashPartitioner`. Однако эти методы можно также выполнять с пользовательским объектом `Partitioner` или объектом `RangePartitioner`.

Пользовательское секционирование

Spark позволяет пользователям описывать собственные объекты `Partitioner` с целью задать специфические функции секционирования данных. Для определения объекта `Partitioner` нужно реализовать методы, указанные ниже.

- `numPartitions` — метод, возвращающий количество секций. Предполагается, что это значение больше нуля.
- `getPartition` — метод, принимающий на входе ключ (того же типа, что в секционируемом наборе RDD) и возвращающий целое число: индекс секции, в которой должны находиться записи с этим ключом. Данному числу следует находиться в диапазоне от нуля до количества секций (заданном в методе `numPartitions`).
- `equals` — необязательный метод для определения равенства объектов `Partitioner`. Для объекта `HashPartitioner` возвращает `true` в случае одинакового количества секций, а для объекта `RangePartitioner` — только в случае совпадения границ диапазонов. Равенство объектов `Partitioner` имеет особенное значение для соединений и совместных группировок: в некоторых случаях, когда согласно объекту `Partitioner` набор RDD уже секционирован, фреймворк Spark

оказывается достаточно «сообразителен» для того, чтобы не перетасовывать его снова таким же образом. Мы обсудим это подробнее в подразделе «Использование совместно расположенных и секционированных наборов RDD» на с. 171.

- `hashcode` — этот метод необходим только в случае переопределения метода `equals`. Хеш-код объекта `HashPartitioner` равен просто количеству секций. Хеш-код объекта `RangePartitioner` представляет собой хеш-функцию, производную от границ диапазонов.

Сохранение информации о секционировании от преобразования к преобразованию

Некоторые широкие преобразования меняют секционирование набора RDD, как будет показано в табл. 6.4. Фреймворк Spark запоминает эту информацию, обновляя свойство `Partitioner` набора RDD. При выполнении серии преобразований важно знать, как секционирован набор RDD, поскольку в некоторых случаях эта информация поможет избежать лишних перетасовок.

Использование сохраняющих секционирование узких преобразований. Некоторые из узких преобразований, например `mapValues`, сохраняют секционирование RDD при его наличии. Если заранее известно, что преобразование меняет в парах «ключ — значение» лишь значение, то у получившегося в его результате набора RDD не будет информации о секционировании (даже если оно не поменялось). Важно отметить: часто используемые преобразования, например `map` и `flatMap`, могут менять и ключ, поэтому, даже если ваша функция этого не делает, у итогового набора RDD не будет известного объекта `Partitioner`. А в ситуациях, когда менять ключи нежелательно, можно задействовать функцию `mapValues` (определенную только для RDD пар «ключ — значение»), поскольку она сохраняет ключи неизменными, а следовательно, объект `Partitioner` тоже не меняется. Функция `mapPartitions` также сохраняет секционирование, если значение флага `preservesPartitioning` равно `true`. Представим, что `data` — набор RDD типа `RDD[(Double, Int)]`, и напишем показанный в примере 6.7 тест для иллюстрации вышесказанного.

Пример 6.7. Сохранение информации о секционировании с помощью функции mapValues

```
val sortedData = data.sortByKey()
val mapValues: RDD[(Double, String)] = sortedData.mapValues(_.toString)
assert(mapValues.partitioner.isDefined,
    "При использовании функции MapValues секционирование сохраняется")

val map = sortedData.map( pair => (pair._1, pair._2.toString))
assert(map.partitioner.isEmpty, "При применении функции map секционирование
    не сохраняется")
```

Использование совместно расположенных и секционированных наборов RDD

Совместно расположенными (co-located) называются наборы RDD с одинаковым объектом `Partitioner`, размещенные на одном физическом участке оперативной памяти. Это важно потому, что все функции класса `CoGroupedRDD` — категория, включающая операции `cogroup` и все операции соединений, — требуют совместного расположения секций группируемых наборов. Объединение наборов RDD без передачи данных по сети возможно только при наличии у них одинаковых объектов `Partitioner` и если все соответствующие секции в оперативной памяти находятся в одном исполнителе. Секции в оперативной памяти окажутся в одном исполнителе при условии, что были секционированы в рамках относящегося к одному заданию графа происхождения.

Совместное секционирование (co-partitioning) — понятие, родственное описанному в предыдущем абзаце, но отличающееся от него. Несколько наборов RDD называются *совместно секционированными*, если секционированы на основе одного и того же известного объекта `Partitioner`.

Две секции называются *совместно расположенными* (co-located), будучи загруженными в оперативную память одной машины. Два набора RDD заведомо совместно расположены, если их поместили в оперативную память одно задание и один объект `Partitioner`: когда секционирование обоих наборов присутствует в графе происхождения одного действия. Наборы RDD будут совместно секционированными при условии одинаковости их объектов `Partitioner`, даже если соответствующие секции каждого из них находятся на разных физических участках оперативной памяти. Напомним, что фраза «одинаковые объекты `Partitioner`» означает их равенство в соответствии с функцией равенства, определенной в классе `Partitioner`.

В примере 6.8 наборы `RDDA` и `RDDB` совместно расположены.

Пример 6.8. Совместно расположенные наборы RDD

```
val RDDA = a.partitionBy(partitionerX)
rddA.cache()
val RDDB = b.partitionBy(partitionerY)
rddB.cache()
val RDDC = a.cogroup(b)
rddC.count()
```

Вследствие отложенной природы вычисления фреймворка Spark ни один из наборов RDD не загружается в оперативную память до момента выполнения `RDDC.count()`. Когда Spark запускает связанное с `RDDC.count()` задание, оба набора RDD подтягиваются в память, так как операция `cogroup` объединила их графы происхождения.

В этом случае соединение не вызовет передачи данных по сети, поскольку оба набора загружены в оперативную память в одном месте.

Напротив, при необходимости вызвать действия над наборами RDDA и RDDB до действия над совместно сгруппированным набором RDD функция cogroup могла бы вызвать определенный сетевой трафик, поскольку наборы не расположены совместно (пример 6.9).

Пример 6.9. Наборы RDD совместно секционированы, но не совместно расположены

```
val RDDA = a.partitionBy(partitionerX)
rddA.cache()
val RDDB = b.partitionBy(partitionerY)
rddB.cache()
val RDDC = a.cogroup(b)
rddA.count()
rddB.count()
rddC.count()
```

В данном случае наборы RDDA и RDDB загружаются в оперативную память различными действиями (см. табл. 6.4). Они совместно секционированы, но их секции не обязательно совместно расположены. Следовательно, хотя повторный вызов функции секционирования и предотвращает перетасовку обоих наборов RDD оператором соединения, вполне возможно появление некоторого трафика из-за выравнивания секций и загрузки обоих наборов в оперативную память. Хотя архитектура программы может потребовать вызова действий именно в этом порядке, часто имеет смысл задуматься над графом происхождения набора RDD, прежде чем вызывать действие над ним с целью минимизировать сетевой трафик.

Перечень функций отображения и секционирования класса PairRDDFunctions

Помимо предназначенных для наборов RDD пар «ключ — значение» функций выборки и отображения, имеются также функции секционирования, приведенные в табл. 6.5.

Класс PairRDDFunctions также включает все операции с несколькими наборами RDD, в том числе join и cogroup, которые мы рассмотрели подробно в главе 4.

Перечень функций класса OrderedRDDOperations

Помимо специальных функций для работы с наборами RDD пар «ключ — значение», существует множество дополнительных функций для наборов RDD с упорядочением ключей, описанных в табл. 6.6.

Таблица 6.4. Перечень операций отображения и выборки для наборов RDD пар «ключ — значение»

Функция	Назначение	Ограничения на ключи	Когда выходит за пределы памяти	Когда начинает работать медленно	Выходной объект Partitioner
mapValues	Применяет функцию отображения ко всем значениям набора RDD пар «ключ — значение» без изменения ключей	Никаких	Практически никогда	Когда операция отображения очень затратна или расширяет размер записей	В отличие от операции map сохраняет секционирование данных для использования в последующих операциях. Если объект Partitioner входного набора RDD известен, то у выходного набора будет тот же Partitioner. При возможности осуществить операцию отображения только для значений почти всегда имеет смысл так и поступить
flatMapValues	Выполняет функцию flatMap только для значений набора RDD пар «ключ — значение». См. обсуждение парадигмы операции flatMap ближе к концу подраздела «Златовласка. Версия 0: итеративное решение» на с. 152	Никаких	Маловероятно, разве что применяемая к каждому из значений функция окажется очень дорогостоящей или итоговый итератор очень велик, что вызовет резкий рост количества записей для каждого ключа	Когда операция отображения очень затратна, создает множество новых объектов или резко увеличивает количество записей (например, при синтаксическом разборе)	Сохраняет объект Partitioner входного набора RDD. Однако распределение дублирующихся значений по ключам может поменяться. При создании большого количества дублирующихся значений это может привести к замедлению дальнейших перетасовок
keys	Возвращает набор RDD из одних ключей (неуникальных)	Никаких	Практически никогда	Почти не подвержена	Сохраняет секционирование
values	Возвращает набор RDD из одних значений	Никаких	Практически никогда	Почти не подвержена	Не сохраняет секционирование. Дальнейшие широкие преобразования приведут к перетасовке даже при таком же объекте Partitioner, как и у входного RDD

Продолжение ⇩

Таблица 6.4 *(продолжение)*

Функция	Назначение	Ограничения на ключи	Когда выходит за пределы памяти	Когда начинает работать медленно	Выходной объект Partitioner
sampleByKey	По заданному ассоциативному словарю с долей в процентах выборки для каждого из ключей возвращает стратифицированную выборку входного набора RDD. Данная функция реализована на основе операции mapPartitions и определяет, где будет храниться та или иная запись с помощью генератора случайных чисел. Следовательно, размер получаемой выборки может лишь приблизительно соответствовать заданным процентным соотношениям	Никаких	Практически никогда, разве что ассоциативный массив ключей слишком велик для трансляции в какой-либо один из рабочих узлов	Аналогично функции mapPartitions эта функция производит один проход по данным и не требует перетасовки	Сохраняет секционирование входных данных

Таблица 6.5. Функции секционирования

Функция	Назначение	Ограничения на ключи	Когда выходит за пределы памяти	Когда начинает работать медленно	Выходной объект Partitioner
partitionBy	Секционирует набор RDD на основе полученного объекта Partitioner. Последний определяет целевые индексы секций для записей на основе ключей	В зависимости от применяемого объекта Partitioner. См. подраздел «Использование объекта Partitioner фреймворка Spark» на с. 167	Когда число дублирующихся значений для каждого ключа слишком велико, независимо от используемого объекта Partitioner	Всегда вызывает перетасовку. Секционирование по диапазонам значений ключей обычно работает медленнее, чем хеш-секционирование, поскольку при этом требуется частичное вычисление данных для осуществления выборки	Секционирован в соответствии с передаваемым аргументом (объектом Partitioner)

Таблица 6.6. Перечень операций из класса OrderedRDDOperations

Функция	Назначение	Ограничения на ключи	Когда выходит за пределы памяти	Когда начинает работать медленно	Выходной объект Partitioner
sortByKey	Возвращает набор RDD, отсортированный по ключу	Ключи должны обладать неявным упорядочением	Когда данные распределены очень неравномерно, как в смысле повторов ключей, так и их распределения. В частности, в случае, когда относящиеся к одному из ключей данные не помещаются в оперативной памяти	Очень много дублирующихся ключей. Если ключи распределены неравномерно, то данные окажутся секционированы тоже неравномерно, что может привести к отстающим задачам. Примечание: это одно из немногих не чистых преобразований, требующих для создания объекта RangePartitioner вызова действия в целях выборки входных данных	Объект RangePartitioner. Количество секций по умолчанию равно количеству секций входного набора RDD
repartitionAndSortWithinPartitions	Получает на входе объект Partitioner и неявное упорядочение. Секционирует набор RDD в соответствии с объектом Partitioner, после чего сортирует все записи в каждой секции в соответствии с неявным упорядочением	См. выше	В зависимости от объекта Partitioner. См. partitionBy	Медленно работает при асимметрии ключей, поскольку секционирование приводит к появлению отстающих задач, а также из-за повышения стоимости сортировки отдельных ключей. Однако эта операция выполняется намного быстрее, чем секционирование с помощью функции partitionBy с последующей сортировкой в пределах каждого из ключей благодаря функции mapPartitions, поскольку вторичная сортировка переносится на шаг перетасовки. См. раздел «Вторичные сортировки и функция repartitionAndSortWithinP3artitions» на с. 176	Такой же, как и аргумент-объект Partitioner
filterByRange	Принимает на входе нижнюю и верхнюю границы для ключей и возвращает набор RDD, состоящий только из тех записей, чьи ключи попадают в этот диапазон	См. выше	Практически никогда	Если набор RDD уже секционирован по диапазонам ключей, то данная операция обходится дешевле, чем обобщенная операция filter, поскольку просматриваются только те секции, ключи которых относятся к нужному диапазону. В противном случае производительность будет такая же, как и у обобщенной операции filter	Сохраняет секционирование

Сортировка по двум ключам с помощью функции SortByKey. Функция `SortByKey` фреймворка Spark позволяет сортировать по кортежам ключей из двух элементов. Так, несложно отсортировать по двум значениям, воспользовавшись неявным упорядочением кортежей языка Scala, после чего отсортировать по составному ключу (`Key1, Key2`). Допустим, что Златовласка захотела, чтобы мы вместо поиска `n`-го элемента создали каталог пар (`индекс столбца, значение`), отсортированных сначала по индексу столбца, а затем по значению. При заданном наборе RDD `indexValuePairs` типа `RDD[(Int, Double)]` эта задача решается следующим образом:

```
indexValuePairs.map((_, null)).sortByKey()
```

Функция `sortByKey` воспользуется неявным упорядочением по кортежам (`Int, Double`), при котором просто сравнивается сначала первое значение из кортежа, а потом — второе.

Функция SortByKey не поддерживает неявное упорядочение по типам произведений, за исключением Tuple2.

Вторичные сортировки и функция repartitionAndSortWithinPartitions

Сортировку в Spark можно осуществить, секционировав набор RDD на основе объекта `RangePartitioner` с последующей сортировкой в пределах каждой секции с помощью функции `mapPartitions`, примерно так, как мы сделали в примере 6.4. Однако такая сортировка выполняется медленнее, чем фактическая реализация функции `SortByKey` из фреймворка Spark. Вместо секционирования с последующей сортировкой фреймворк Spark применяет метод под названием *«вторичная сортировка»* (secondary sort), который делегирует этапу перетасовки задачу по сортировке данных на отдельной машине.

Вторичная сортировка — весьма эффективный способ упорядочения данных как между машинами, так и в пределах одной машины. Данный термин берет свое начало из парадигмы MapReduce и описывает методику, при которой разработчик с помощью одной функции выполняет отображение, задавая при этом для использования при свертке другой порядок элементов. В итоге появляется возможность сделать часть локальной работы по сортировке в Spark на этапе перетасовки, а не на следующем, после завершения перетасовки.

Во фреймворке Spark есть встроенная функция для вторичной сортировки — `repartitionAndSortWithinPartitions`. Она представляет собой широкое преобра-

зование, принимающее на входе объект `Partitioner` — определенный в передаваемом в качестве аргумента наборе RDD — и неявное упорядочение, которое должно быть определено для ключей данного набора. Эта функция секционирует данные в соответствии с аргументом `Partitioner`, после чего сортирует записи в каждой из секций в соответствии с установленным упорядочением.

Нет нужды непосредственно передавать неявное упорядочение в функцию repartitionAndSortWithinPartitions. Если ее вызывают для набора RDD с упорядоченными ключами, то фреймворк Spark сам сделает вывод относительно упорядочения и отсортирует данные соответствующим образом. Чтобы использовать функцию repartitionAndSortWithinPartitions для сортировки типов, у которых или нет неявного упорядочения, или оно не такое, как нужно, необходимо описать неявное упорядочение для ключей набора RDD в программе до ее вызова. См. пример 6.12.

Если заглянуть в реализацию `sortByKey`, то можно увидеть, что она вызывает функцию `repartitionAndSortWithinPartitions` с параметром `RangePartitioner` и использует заданное для ключей неявное упорядочение. Как мы уже обсуждали в подразделе «Секционирование по диапазонам значений ключей» на с. 168, объект `RangePartitioner` осуществляет выборку данных и ставит каждой из секций в соответствие диапазон значений на основе выведенного им распределения ключей (например, ключи со значениями между 0 и 10 должны быть помещены в секцию с индексом 2). После этого функция `repartitionAndSortWithinPartitions` сортирует значения в каждой из секций (в каждом диапазоне данных) и таким образом весь итоговый результат оказывается отсортирован по ключу. Парадигма вторичной сортировки и функция `repartitionAndSortWithinPartitions` могут использоваться не только для эффективной сортировки по одному ключу, но и для установки двух видов упорядочения данных: одного для секционирования и второго для упорядочения элементов дочерних секций. Оставшаяся часть данного раздела будет посвящена второму из этих сценариев применения, при котором требуется упорядочить данные сначала по одному критерию, а затем — по другому.

Использование repartitionAndSortWithinPartitions для функции группировки по ключу с сортировкой значений

Оптимальный способ упорядочить данные по двум критериям — воспользоваться функцией `repartitionAndSortWithinPartitions`. Один из частых сценариев применения для нее — описание функции, которую можно назвать `groupByKeyAndSortValues` (группировка по ключу с сортировкой значений), возвращающей сгруппированный по ключу набор RDD, значения в каждой из групп которого отсортированы. В отличие от сортировки по ключам-кортежам, обсуждавшейся в пункте

«Сортировка по двум ключам с помощью функции SortByKey» на с. 176, этот подход можно обобщить на случай любой схемы секционирования при любом типе ключей и пользовательском упорядочении. Мы задействуем функцию `repartitionAndSortWithinPartitions` для повторного секционирования набора RDD по одному упорядочению ключей и затем определим неявное упорядочение для записей в заданной секции.

Немного поискав, вы легко найдете множество функций `groupByKeyAndSortValues`, хотя ни одна из них не была включена во фреймворк Spark. Особенно хорош вариант Сэнди Ризы в *«Spark для профессионалов»*[1], и приведенная далее реализация очень напоминает этот вариант. Указанная функция предполагает данные в виде `((k, s), v)`, где `s` — вторичный ключ (возможно, производный от значений). Она секционирует набор RDD по первой части ключа, а затем сортирует по второй. Далее функция объединяет все относящиеся к одному ключу значения в отсортированный итератор.

Функция `groupByKeyAndSortValues` выполняется в четыре шага.

1. Определение пользовательского объекта `Partitioner`, секционирующего записи в соответствии с первым элементом ключа.
2. Определение неявного упорядочения для значений. Это требуется лишь потому, что функция является обобщенной. Неявное упорядочение для кортежей: первое значение, второе значение. Нужно лишь указать фреймворку Spark на необходимость использования данного упорядочения кортежей.
3. Применение ко входному набору RDD функции `repartitionAndSortWithinPartitions` с пользовательским объектом `Partitioner`, определенным в шаге 1.
4. Объединение элементов с помощью операции `mapPartitions`. Можно опереться на тот факт, что элементы с одинаковым первым ключом находятся в одной секции, а элементы в пределах каждой секции отсортированы сначала в соответствии с первым упорядочением, а затем — со вторым.

На самом деле данная функция не сортирует значения по первому ключу, поскольку используется хеш-секционирование. Вместо этого она группирует ключи с одинаковым хеш-значением, размещая их на одной машине. Таким образом, если мы выполним ее для значений от 1 до 5 при четырех секциях, то первая секция будет содержать значения 1 и 5. Чтобы ключи действительно сортировались, необходимо задать объект RangePartitioner. Однако если наша задача — просто сгруппировать схожие ключи, то использования хеш-значения вполне достаточно.

[1] Spark для профессионалов: современные паттерны обработки больших данных / С. Риза, У. Лезерсон, Ш. Оуэн, Д. Уиллс. — СПб. : Питер, 2017. — 272 с. — ISBN 978-5-496-02401-3.

В примере 6.10 приведен код пользовательского объекта `Partitioner`. Как вы можете видеть, мы упорядочиваем только первую часть ключа. Если задать упорядочение по обеим частям ключа, то группировка будет выполняться по хеш-значению всего кортежа ключа. Следовательно, функция может поместить элементы с одинаковым первичным ключом, но разными вторичными, в две разные секции.

Пример 6.10. Шаг 1 вторичной сортировки: определение пользовательского объекта Partitioner

```
class PrimaryKeyPartitioner[K, S](partitions: Int) extends Partitioner {
  /**
   * Создаем объект HashPartitioner и используем его с первым набором ключей
   */
  val delegatePartitioner = new HashPartitioner(partitions)

  override def numPartitions = delegatePartitioner.numPartitions
  /**
   * Секционируем по хеш-значению первого ключа
   */
  override def getPartition(key: Any): Int = {
    val k = key.asInstanceOf[(K, S)]
    delegatePartitioner.getPartition(k._1)
  }
}
```

Далее необходимо задать неявное упорядочение, как показано в примере 6.11. Как вы помните из определения функции, для обеих частей ключа упорядочение уже задано. А во фреймворке Spark уже существует упорядочение для кортежей из двух упорядоченных элементов, следовательно, осталось просто указать Spark, что нужно использовать обобщенное упорядочение Tuple2.

Пример 6.11. Шаг 2 вторичной сортировки: определение неявного упорядочения

```
implicit def orderByLocationAndName[A <: PandaKey]: Ordering[A] = {
  Ordering.by(pandaKey => (pandaKey.city, pandaKey.zip, pandaKey.name))
}

implicit val ordering: Ordering[(K, S)] = Ordering.Tuple2
```

Теперь, создав эти две подпрограммы, можно описать функцию `groupByKeyAndSortBySecondaryKey`, как показано в примере 6.12. Эта новая функция секционирует данные в соответствии с описанным в шаге 1 объектом `Partitioner`, сортируя их в соответствии с описанным в шаге 2 упорядочением, после чего с помощью функции `groupSorted` объединяет элементы с одинаковым первым ключом в один итератор.

Пример 6.12. Общий образец функции для «группировки по ключу с сортировкой по вторичному ключу»

```
def groupByKeyAndSortBySecondaryKey[K : Ordering : ClassTag,
  S : Ordering : ClassTag,
  V : ClassTag]
```

```
   (pairRDD : RDD[((K, S), V)], partitions : Int):
    RDD[(K, List[(S, V)])] = {
  // Создаем экземпляр нашего пользовательского Partitioner
  val colValuePartitioner = new PrimaryKeyPartitioner[Double, Int](partitions)

  // Задаем неявное упорядочение для сортировки по второму ключу,
  // которое будет использоваться даже без явного обращения
  implicit val ordering: Ordering[(K, S)] = Ordering.Tuple2

  // Используем функцию repartitionAndSortWithinPartitions
  val sortedWithinParts =
    pairRDD.repartitionAndSortWithinPartitions(colValuePartitioner)

  sortedWithinParts.mapPartitions( iter => groupSorted[K, S, V](iter) )
}

def groupSorted[K,S,V](
  it: Iterator[((K, S), V)]): Iterator[(K, List[(S, V)])] = {
  val res = List[(K, ArrayBuffer[(S, V)])]()
  it.foldLeft(res)((list, next) => list match {
    case Nil =>
      val ((firstKey, secondKey), value) = next
      List((firstKey, ArrayBuffer((secondKey, value))))

    case head :: rest =>
      val (curKey, valueBuf) = head
      val ((firstKey, secondKey), value) = next
      if (!firstKey.equals(curKey) ) {
         (firstKey, ArrayBuffer((secondKey, value))) :: list
      } else {
        valueBuf.append((secondKey, value))
        list
      }

  }).map { case (key, buf) => (key, buf.toList) }.iterator
}
```

При разработке подобной функции, существенно зависящей от секционирования, обязательно убедитесь, что ваши модульные тесты подходят для данных, занимающих более одной секции. Обязательно протестируйте при различном количестве секций и различных данных, поскольку секционирование является не вполне детерминистичным (особенно секционирование по диапазонам значений ключей). Больше информации о том, как создать хорошие распределенные тесты, вы можете найти в главе 8.

Как не следует сортировать по двум критериям

Важно отметить, что некоторые из других, кажущихся очевидными, подходов к решению этой задачи совсем не обязательно дадут верный результат. Например, даже если не упоминать о производительности, функция `groupByKey` не обеспечи-

вает порядок значений в группах. Таким образом, следующая реализация может возвращать неправильные результаты:

```
indexValuePairs.sortByKey().groupByKey()
```

Не гарантируется и устойчивость сортировки Spark (сохранение исходной упорядоченности элементов при том же значении). Поэтому повтор сортировки неприемлем:

```
indexValuePairs.sortByKey.map(_.swap()).sortByKey
```

В этом случае второй вызов функции `sortByKey` может не сохранять полученное при первой сортировке упорядочение.

Златовласка. Версия 2: вторичная сортировка

Логика вторичной сортировки распространяется далеко за пределы простого упорядочения данных. Она применима к любым сценариям использования, требующим организации записей в соответствии с двумя различными ключами. Первоначальный пример со Златовлаской связан со вторичной сортировкой, поскольку требует перетасовки по одному из ключей (по индексу столбца) с последующим упорядочением данных по значениям внутри каждого из ключей (по значениям в ячейках). Следовательно, вместо того, чтобы применять функцию `groupByKey` для объединения значений, относящихся к каждому из ключей, с последующей сортировкой на отдельном шаге элементов, относящихся к каждому из ключей, можно задействовать функцию `repartitionAndSortWithinPartitions`. Она позволяет выполнить секционирование по индексу столбца и отсортировать по значениям каждого из столбцов. Это гарантирует, что все относящиеся к каждому из столбцов значения будут располагаться в одной секции, причем в упорядоченном виде. Затем можно просто пройти в цикле по элементам всех секций, отфильтровать нужные ранговые статистики за один проход по данным и воспользоваться функцией `groupSorted` для объединения относящихся к нужному столбцу ранговых статистик.

Определение пользовательского объекта Partitioner

Упорядочение и секционирование в функции `repartitionAndSortWithinPartitions` должно производиться по ключам набора RDD, следовательно, необходимо применять в качестве ключей пары (индекс столбца, значение). Можно задействовать фиктивное значение (скажем, `1` или `null`), чтобы Spark интерпретировал набор RDD как набор пар «ключ — значение», где ключи представляют собой кортежи (`индекс столбца, значение`). Далее нужно будет задать пользовательский объект `Partitioner` для секционирования ключей по хеш-значению первой части ключа (индексу столбца), как показано в примере 6.13.

Пример 6.13. Златовласка. Версия 2, определение пользовательского объекта Partitioner для секционирования по индексу столбца

```
class ColumnIndexPartition(override val numPartitions: Int)
  extends Partitioner {
  require(numPartitions >= 0, s"Число секций " +
    s"($numPartitions) не может быть меньше нуля.")

  override def getPartition(key: Any): Int = {
    val k = key.asInstanceOf[(Int, Double)]
    Math.abs(k._1)  % numPartitions // Хеш-код индекса столбца
  }
}
```

Фильтрация по секциям

Элементы в каждой из секций нужно упорядочить сначала по индексу столбца, а затем — по значению. Первое гарантирует, что все значения, относящиеся к одному ключу, находятся в одной секции. Второе — что у всех соседних элементов будут одинаковые индексы столбцов, отсортированные так, что можно вычислить ранговую статистику.

Существующее неявное упорядочение для кортежей — сначала по первому значению из кортежа, а затем — по второму. Следовательно, не нужно задавать для данных упорядочение. После вызова функции `repartitionAndSortWithinPartitions` они окажутся секционированы в соответствии с индексом столбца и отсортированы по индексу столбца и значению. Например, мы используем описанный в табл. 6.1 объект `DataFrame` и количество секций, равное трем. Первая секция должна содержать следующие данные:

```
((1, 2.0), 1)
((1, 3.0), 1)
((1, 10.0), 1)
((1, 15.0), 1)
((4, 0.0), 1)
((4, 0.0), 1)
((4, 0.0), 1)
((4, 98.0), 1)
```

Можно воспользоваться операцией `filter` для прохода в цикле по элементам итератора, несмотря на то что фильтрация требует отслеживания глобальных данных. Как вы помните из нашего обсуждения преобразований «итератор — итератор» в разделе «Выполнение преобразований “итератор — итератор” с помощью функции mapPartitions» на с. 121, операции над итераторами `map`, `filter` и `flatMap` преобразуют элементы итератора по очереди. Следовательно, поскольку элементы отсортированы и сгруппированы по ключу, можно вести промежуточные суммы по индексам столбцов. Если элемент — один из относящихся к целевой ранговой статистике, то мы его оставляем. После этого нужно будет отобразить итератор на

первую половину кортежа, чтобы удалить фиктивное значение 1. Обратите внимание: можно объединить шаги отображения и фильтрации в одну операцию flatMap. Мы решили разделить их в примере 6.14, поскольку полагаем, что операция filter более наглядна.

Пример 6.14. Златовласка. Версия 2, использование функции repartitionAndSortWithinPartitions

```
def findRankStatistics(dataFrame: DataFrame,
  targetRanks: List[Long], partitions: Int) = {

  val pairRDD: RDD[((Int, Double), Int)] =
    GoldilocksGroupByKey.mapToKeyValuePairs(dataFrame).map((_, 1))

  val partitioner = new ColumnIndexPartition(partitions)
   // Сортируем на основе существующего неявного упорядочения:
   // по первому ключу кортежа, по второму ключу кортежа
  val sorted = pairRDD.repartitionAndSortWithinPartitions(partitioner)

  // Отфильтровываем нужные ранговые статистики
  val filterForTargetIndex: RDD[(Int, Double)] =
    sorted.mapPartitions(iter => {
      var currentColumnIndex = -1
      var runningTotal = 0
      iter.filter({
        case (((colIndex, value), _)) =>
          if (colIndex != currentColumnIndex) {
            currentColumnIndex = colIndex // Заменяем новым индексом столбца
            runningTotal = 1
          } else {
            runningTotal += 1
          }
        // Если промежуточная сумма относится к одной из нужных
        // ранговых статистик, сохраняем данную пару ((colIndex, value))
        targetRanks.contains(runningTotal)
    })
  }.map(_._1), preservesPartitioning = true)
  groupSorted(filterForTargetIndex.collect())
}
```

Объединяем относящиеся к одному ключу элементы

После шага mapPartitions остается выполнить одно локальное преобразование, чтобы группировать в ассоциативный массив элементы, относящиеся к одному индексу столбца. Код предназначенной для этой цели функции groupSorted приведен в примере 6.15.

Пример 6.15. Златовласка. Версия 2, группировка относящихся к одному ключу элементов

```
private def groupSorted(
  it: Array[(Int, Double)]): Map[Int, Iterable[Double]] = {
```

```
  val res = List[(Int, ArrayBuffer[Double])]()
  it.foldLeft(res)((list, next) => list match {
    case Nil =>
      val (firstKey, value) = next
      List((firstKey, ArrayBuffer(value)))
    case head :: rest =>
      val (curKey, valueBuf) = head
      val (firstKey, value) = next
      if (!firstKey.equals(curKey)) {
         (firstKey, ArrayBuffer(value)) :: list
      } else {
        valueBuf.append(value)
        list
      }
  }).map { case (key, buf) => (key, buf.toIterable) }.toMap
}
```

Обратите внимание: этот код очень напоминает функцию группировки, приведенную в примере 6.12.

Производительность

Быстродействие этого решения намного выше на любых данных, чем версии 1 функции `groupByKey`. Применение функции `repartitionAndSortWithinPartitions` позволило делегировать этапу перетасовки работу по сортировке столбцов. А так как элементы сортируются после перетасовки, появляется возможность использовать преобразования «итератор — итератор» для фильтрации данных и избежать помещения всех относящихся к секции данных в оперативную память.

Однако если столбцы относительно «длинные», то шаг `repartitionAndSortWithinPartitions` может привести к сбоям, поскольку для него требуется наличие в одном исполнителе достаточного места для хранения всех значений, относящихся к столбцам с одинаковым хеш-значением. И действительно, в нашем случае на этапе перетасовки по-прежнему происходят сбои при использовании этого подхода для значительных объемов данных. На самом деле приемлемое решение задачи Златовласки требует совершенно иного подхода.

Другой подход к задаче Златовласки

К сожалению, ни одно из существующих преобразований данных типа «ключ — значение» не решает, как по мановению волшебной палочки, проблемы Златовласки. Ни одна из альтернативных `groupByKey` агрегирующих операций не подойдет, поскольку операция, которую нужно выполнить для каждого из ключей, — сортировка — не уменьшает размера соответствующих этому ключу данных. Как мы

обсуждали в предыдущем подразделе, даже переписывание нашего основанного на функции `groupByKey` решения с использованием усовершенствованных методик вторичной сортировки приводит к сбоям. В конце концов подход со вторичной сортировкой по-прежнему требует секционирования по индексу столбца, недостаточно «мелкозернистого» для имеющегося объема данных и доступных ресурсов. На самом деле эффективное решение этой задачи требует полного переосмысления способа распараллеливания вычислений.

Прежде чем углубиться в детали этого решения, перечислим некоторые из изученных методов повышения эффективности преобразований.

- Узкие преобразования над данными типа «ключ — значение» быстрее и легче распараллеливаются по сравнению с требующими перетасовки широкими преобразованиями.
- Некоторые узкие преобразования позволяют сохранить локальность секций после перетасовки. Это относится к операции `mapPartitions` при заданной опции `preservesPartitioning=true` и к `mapValues`.
- Широкие преобразования оптимальны в случае большого количества неповторяющихся ключей. Это предотвращает размещение большой доли данных в одном исполнителе в результате перетасовки.
- Операция `sortByKey` — особенно хороший способ секционирования данных и сортировки внутри секций, поскольку делегирует упорядочение данных на локальных машинах этапу перетасовки.
- Использование преобразований «итератор — итератор» в `mapPartitions` предотвращает загрузку целых секций в оперативную память.
- Иногда можно воспользоваться перетасовочными файлами для предотвращения повторных вычислений широких преобразований, даже при вызове нескольких действий над одним набором RDD.

Благодаря этим наблюдениям можно создать решение для задачи Златовласки на основе всего лишь одной операции `sortByKey` и трех операций `mapPartitions`. Самый важный вывод: совсем не обязательно, чтобы единицей параллелизма для данной задачи был столбец. По существу, задача решаема для каждого из диапазонов значений. Если значения в ячейках отсортированы и мы знаем, сколько элементов из каждого столбца содержится в каждой из секций (что легко подсчитать с помощью эффективной процедуры `mapPartitions`), то можно просто определить расположение `n`-го элемента.

Наше решение можно разбить на пять шагов.

1. Отобразить строки данных в виде пар `(значение ячейки, индекс)`.
2. Выполнить операцию `sortByKey` для всех кортежей, определенных на шаге 1.

3. Определить с помощью операции `mapPartitions`, сколько элементов из каждого столбца содержится в каждой из секций, и отправить эту информацию на драйвер путем операции `collect`.
4. Произвести над результатами шага 3 локальное вычисление, чтобы определить местоположение всех требуемых ранговых статистик. Так, допустим, мы хотим найти 13-й элемент. Допустим также, что на шаге 3 мы выяснили: в первой секции содержатся десять элементов из столбца 6. В этом случае можем заключить, что 13-й элемент будет третьим по величине элементом столбца 6 во второй секции.
5. Выделить путем фильтрации элементы, соответствующие нужным ранговым статистикам, с помощью еще одного преобразования `mapPartitions` с использованием результатов, полученных на шаге 4.

Отображение строк данных в виде пар (значение ячейки, индекс столбца)

Пример 6.16 представляет собой код шага 1 нашего решения: отображение строк данных в виде пар (`значение ячейки, индекс столбца`). Мы воспользуемся функцией `flatMap` фреймворка Spark, чтобы преобразовать каждую из строк в последовательность кортежей.

Пример 6.16. Алгоритм решения задачи Златовласки. Версия 3, отображение в виде пар (значение ячейки, индекс столбца)

```
private def getValueColumnPairs(dataFrame : DataFrame): RDD[(Double, Int)] = {
  dataFrame.rdd.flatMap{
    row: Row => row.toSeq.zipWithIndex
                .map{
                  case (v, index) => (v.toString.toDouble, index)}
  }
}
```

Сортировка и подсчет значений в каждой из секций

Отобразив строки таким образом, чтобы ключами служили значения ячеек, воспользуемся операцией `sortByKey`. После сортировки подсчитаем количество элементов в каждой из секций. Пример 6.17 представляет собой функцию, принимающую на входе отсортированный набор RDD пар вида (`значение типа double, индекс столбца`) и количество столбцов исходного набора `DataFrame` и возвращающую массив, в котором каждый элемент соответствует секции. Каждый элемент данного массива содержит индекс секции и массив с количеством элементов в каждом из столбцов этой секции. Длина каждого из подмассивов соответствует количеству столбцов в исходном наборе данных.

Пример 6.17. Алгоритм решения задачи Златовласки. Версия 3, подсчет значений по столбцам в каждой из секций

```
private def getColumnsFreqPerPartition(sortedValueColumnPairs: RDD[(Double, Int)],
  numOfColumns : Int):
  Array[(Int, Array[Long])] = {

  val zero = Array.fill[Long](numOfColumns)(0)

  def aggregateColumnFrequencies (partitionIndex : Int,
    valueColumnPairs : Iterator[(Double, Int)]) = {
    val columnsFreq : Array[Long] = valueColumnPairs.aggregate(zero)(
       (a : Array[Long], v : (Double, Int)) => {
        val (value, colIndex) = v
        // Увеличиваем значение ячейки в массиве zero,
        // соответствующей данному индексу столбца
        a(colIndex) = a(colIndex) + 1L
        a
      },
       (a : Array[Long], b : Array[Long]) => {
        a.zip(b).map{ case(aVal, bVal) => aVal + bVal}
      })

    Iterator((partitionIndex, columnsFreq))
  }

  sortedValueColumnPairs.mapPartitionsWithIndex(
    aggregateColumnFrequencies).collect()
}
```

Подфункция `aggregateColumnFrequencies` применяется к записям во всех секциях. В ней используется операция агрегирования, определенная для итераторов. Переменная `zero` — массив длины, соответствующей исходному количеству столбцов. Для каждой пары в итераторе операция формирования последовательности операции агрегирования наращивает значение ячейки, соответствующей данному индексу столбца в массиве `zero`. Операция объединения складывает значения из этих двух массивов. Таким образом, результатом является массив счетчиков для соответствующих индексов столбцов. Например, представим, что у нас есть три столбца, причем первые две секции содержат следующие пары «ключ — значение»:

```
Partition 1: (1.5, 0) (1.25, 1) (2.0, 2) (5.25, 0)
Partition 2: (7.5, 1) (9.5, 2)
```

Результат будет следующим:

```
[(0, [2, 1, 1]), (1, [0, 1, 1])]
```

Вычислительные затраты на этот шаг должны быть относительно невысокими. Шаг `mapPartitions` представляет собой узкое преобразование, поскольку требует однократного прохода итератора. Следовательно, операция не вызовет перетасовки и может сбрасывать данные на диск избирательно. Мы применим массивы

для агрегирования, поскольку (как мы обсуждали в подразделе «Использование меньших структур данных» на с. 118) при них расходы на сборку мусора будут минимальными. После этого шага `mapPartitions` соберем ее результаты в массив.

Мы отправляем результаты операции mapPartitions на драйвер с помощью вызова функции collect, а значит, не используем их распределенным образом. Следовательно, не нужно устанавливать значение флага preservesPartitioning в false.

Определение местоположения ранговых статистик в каждой из секций

После получения результатов функции `getColumnsFreqPerPartition` необходимо на их основе определить, где находятся ранговые статистики в каждой из секций, как показано в примере 6.18. Это вычисление производится локально. Чтобы определить местоположение всех ранговых статистик, пройдем в цикле по (отсортированным) результатам работы предыдущей функции, ведя параллельно промежуточные суммы элементов в каждом столбце по секциям. Если ранговая статистика оказывается (для какого-либо из столбцов) между предыдущим и обновленным значениями промежуточной суммы, то, значит, она находится в данной секции. В таком случае мы добавляем в `relevantIndexList` индекс столбца и ранговую статистику минус предыдущую промежуточную сумму. Далее можем вернуть массив, в котором каждый элемент будет индексом секции, и объект — список пар. Каждая пара представляет собой индекс столбца для этой ранговой статистики и индекс данной ранговой статистики в столбце.

Например, если входные данные для нашей функции вот такие:

```
targetRanks: [5]
partitionColumnsFreq: [(0, [2, 3]), (1, [4, 1]), (2, [5, 2])]
numOfColumns: 2
```

то результаты будут следующими:

```
(0, []), (1, [(0, 3)]), (2, [(1, 1)])]
```

Пример 6.18. Алгоритм решения задачи Златовласки. Версия 3, определение местоположений ранговых статистик в каждой из секций

```
private def getRanksLocationsWithinEachPart(targetRanks : List[Long],
      partitionColumnsFreq : Array[(Int, Array[Long])],
      numOfColumns : Int) : Array[(Int, List[(Int, Long)])] = {

  val runningTotal = Array.fill[Long](numOfColumns)(0)
  // Индексы секций не обязательно отсортированы, так что придется
  // отсортировать массив partitionsColumnsFreq по индексу секции
  // (первое значение в кортеже)
```

```
  partitionColumnsFreq.sortBy(_._1).map { case (partitionIndex, columnsFreq) =>
    val relevantIndexList = new MutableList[(Int, Long)]()

    columnsFreq.zipWithIndex.foreach{ case (colCount, colIndex)  =>
      val runningTotalCol = runningTotal(colIndex)
      val ranksHere: List[Long] = targetRanks.filter(rank =>
        runningTotalCol < rank && runningTotalCol + colCount >= rank)

      // Для каждой имеющейся ранговой статистики добавляем индекс
      // столбца и индекс ее в этой секции (ранг – промежуточная сумма)
      relevantIndexList ++= ranksHere.map(
        rank => (colIndex, rank - runningTotalCol))

      runningTotal(colIndex) += colCount
    }

     (partitionIndex, relevantIndexList.toList)
  }
}
```

Фильтр для ранговых статистик

Теперь, после выяснения местоположения (индекса секции и позиции внутри секции) ранговых статистик для каждого из столбцов, мы должны передать отсортированные данные в фильтр для получения нужных ранговых статистик. Эта задача решается функцией, представленной в примере 6.19, под названием `findTargetRanksIteratively`, использующей первоначальные отсортированные кортежи (`значение, пары индексов столбцов`) и результаты предыдущей функции. Мы задействуем преобразование «итератор — итератор», включающее шаги `filter` и `map` (обратите внимание: их можно заменить операцией `flatMap`). Оно возвращает итоговый результат, набор RDD-пар (`индекс столбца, ранговая статистика`), который мы затем отправляем обратно на драйвер с помощью операции `collect`.

Пример 6.19. Алгоритм решения задачи Златовласки. Версия 3, фильтрация нужных ранговых статистик

```
private def findTargetRanksIteratively(
  sortedValueColumnPairs : RDD[(Double, Int)],
  ranksLocations : Array[(Int, List[(Int, Long)])]):
    RDD[(Int, Double)] = {

  sortedValueColumnPairs.mapPartitionsWithIndex(
     (partitionIndex : Int, valueColumnPairs : Iterator[(Double, Int)]) => {
      val targetsInThisPart: List[(Int, Long)] = ranksLocations(partitionIndex)._2
      if (targetsInThisPart.nonEmpty) {
        val columnsRelativeIndex: Map[Int, List[Long]] =
        targetsInThisPart.groupBy(_._1).mapValues(_.map(_._2))
        val columnsInThisPart = targetsInThisPart.map(_._1).distinct

        val runningTotals : mutable.HashMap[Int, Long]= new mutable.HashMap()
```

```
        runningTotals ++= columnsInThisPart.map(
          columnIndex => (columnIndex, 0L)).toMap

  // Фильтруем этот итератор,чтобы он содержал только пары (значение, индекс столбца),
  // представляющие собой ранговые статистики для данной секции
  // Отслеживаем количество просмотренных элементов для каждого columnIndex
  // с помощью hashMap с промежуточными суммами
      valueColumnPairs.filter{
        case(value, colIndex) =>
          lazy val thisPairIsTheRankStatistic: Boolean = {
            val total = runningTotals(colIndex) + 1L
            runningTotals.update(colIndex, total)
            columnsRelativeIndex(colIndex).contains(total)
          }
            (runningTotals contains colIndex) && thisPairIsTheRankStatistic
      }.map(_.swap)
    }
    else {
      Iterator.empty
    }
  })
}
```

Златовласка. Версия 3: сортировка по значениям ячеек

Объединяя все эти функции, получаем полное решение задачи Златовласки (пример 6.20).

Пример 6.20. Алгоритм решения задачи Златовласки. Версия 3: сортировка значений

```
def findRankStatistics(dataFrame: DataFrame, targetRanks: List[Long]):
  Map[Int, Iterable[Double]] = {

  val valueColumnPairs: RDD[(Double, Int)] = getValueColumnPairs(dataFrame)
  val sortedValueColumnPairs = valueColumnPairs.sortByKey()
  sortedValueColumnPairs.persist(StorageLevel.MEMORY_AND_DISK)

  val numOfColumns = dataFrame.schema.length
  val partitionColumnsFreq =
    getColumnsFreqPerPartition(sortedValueColumnPairs, numOfColumns)
  val ranksLocations = getRanksLocationsWithinEachPart(
      targetRanks, partitionColumnsFreq, numOfColumns)
  val targetRanksValues = findTargetRanksIteratively(
    sortedValueColumnPairs, ranksLocations)
  targetRanksValues.groupByKey().collectAsMap()
}
```

С точки зрения удобочитаемости кода представленное решение довольно некрасиво. Оно требует десятков строк кода и четырех проходов по данным. Однако можно ожидать, что оно позволит избежать ошибок нехватки памяти на исполнителях и будет работать быстрее решений на основе операции `groupByKey` или вторичной сортировки. Это обусловлено тем, что данные в каждом столбце — в основном неповторяющиеся числа типа `double`, а следовательно, перетасовка станет выполняться весьма эффективно. Последние две процедуры `mapPartitions` вклю-

чают свертку данных и могут осуществляться с помощью преобразований «итератор — итератор», вследствие чего можно ожидать, что они будут масштабироваться достаточно хорошо. Несомненно, на случайно распределенных тестовых данных это решение по производительности будет на порядок превосходить другие решения задачи Златовласки.

Выявление отстающих задач и несбалансированных данных

Отстающими называются задачи, выполнение которых занимает намного больше времени по сравнению с другими задачами того же этапа. Как вы помните из нашего обсуждения в разделе «Планирование заданий Spark» на с. 38, новый этап начинается по завершении широкого преобразования. При вызове подобных преобразований для одного и того же набора RDD этапы обычно выполняются последовательно, так что отстающие задачи могут приводить к задержке всего задания. Они возникают при неправильном выделении ресурсов фреймворком Spark, в частности, когда данные распределены неравномерно. Отстающие задачи — верный признак несбалансированных ключей, так как распределение задач зависит от секционирования, которое, в свою очередь, зависит от распределения ключей. Веб-интерфейс Spark позволяет мониторить выполнение задач в режиме реального времени.

Если во время широкого преобразования вы заметили, что вычисление некоторых секций занимает значительно больше времени, чем остальных, или у них намного выше процент повторных вычислений, то весьма вероятно неравномерное распределение данных. Оно обычно происходит из-за того, что количество значений для одних ключей гораздо больше, чем для других. В таком случае ускорить операции перетасовки можно или использовав что-либо другое в качестве ключей, или добавив в ключи случайный шум с целью повысить степень их уникальности. Иногда можно даже выполнить свертку при отображении, чтобы объединить или отфильтровать записи с дубликатами в каждой из секций до перетасовки всех данных.

Хотя при больших масштабах вычислений вероятность ошибок памяти при операции sortByKey ниже, чем при groupByKey, они не исключены полностью. Взгляните снова на рис. 6.2 — и увидите, что «перебор» памяти при операции sortByKey на рис. 6.4 по-прежнему возможен.

Один из обходных путей решения проблемы несбалансированных ключей — добавление какого-нибудь мусора в конец ключа, например случайного числа. Таким образом, Spark будет считать ключи различными и распределять их по секциям. В нашем случае это может означать распределение нулей по машинам, что никак не должно повлиять на точность.

Следующие рисунки иллюстрируют несбалансированную (рис. 6.4) и сбалансированную (рис. 6.5) операции `sortByKey`.

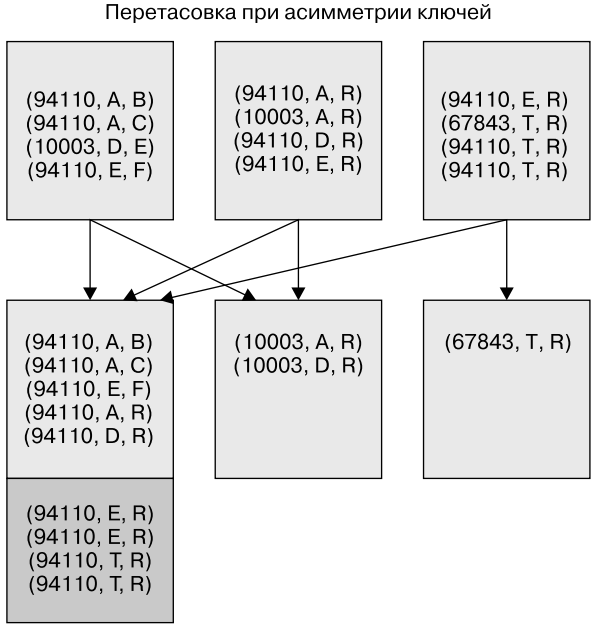

Рис. 6.4. Ошибки памяти при операции sortByKey

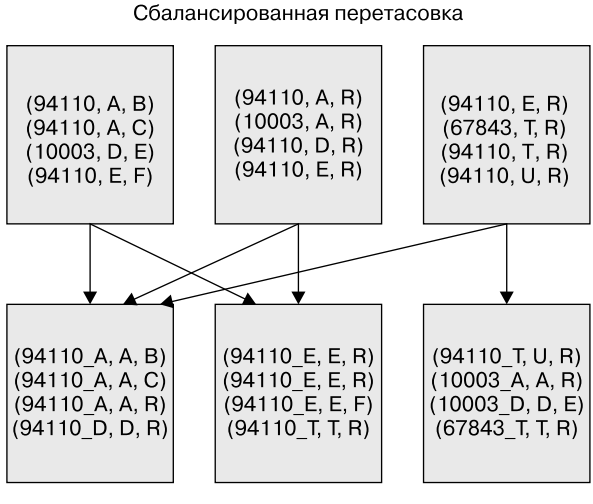

Рис. 6.5. Сбалансированная перетасовка при операции sortByKey

Возвращаемся к Златовласке

Златовласка заверила нас, что ее данные распределены равномерно, а во всех столбцах множество различных значений. Однако попытка запустить алгоритм на ее данных провалилась. Посмотрев на его работу в веб-интерфейсе, мы обратили внимание, что одни секции обрабатывались намного медленнее других и иногда в них происходили выходы за пределы памяти. Это указывает на слишком большое количество дублирующихся ключей. Странная ситуация, ведь в нашей реализации ключами были значения данных (числа типа `double`). Однако дальнейший анализ показал: в большинстве столбцов около 25 % значений были нулевыми. Следовательно, приблизительно каждый четвертый ключ в нашей сортировке был нулем. Это значит, что 1/4 записей в сортируемом наборе RDD сконцентрировалась возле нуля. Соответственно, вне зависимости от количества используемых секций четверть записей оказалась при перетасовке в первых нескольких секциях.

Златовласка. Версия 4: свертка до уникальных значений по каждой секции

Вместо того чтобы пытаться секционировать эти пары по-разному, можно модифицировать последние четыре шага алгоритма так, чтобы они имели дело с входными данными в виде кортежей (`(значение ячейки, индекс столбца), количество`). Далее, на первом шаге вместо отображения записей в каждой из секций в пары (`значение ячейки, индекс столбца`) можно отобразить каждую секцию в уникальные пары, отслеживая попутно количество вхождений каждой пары (`(значение ячейки, индекс столбца)`) в этой секции. С помощью отображения в уникальные значения в каждой из секций мы снижаем количество дублирующихся ключей без перетасовки (в отличие от распределенного вызова `distinct` для всего набора RDD).

Сформировав эти кортежи (`(значение ячейки, индекс столбца), количество`), мы знаем: хотя некоторое количество повторяющихся ключей возможно, мы установили теоретический предел для количества дублирующихся ключей. Точнее говоря, если одно и то же значение присутствует в столбце в каждой из секций, то максимальное количество дублирующихся ключей составляет количество столбцов × количество секций[1]. Такой исходный шаг не только выравнивает данные, позволяя выполнять более эффективное секционирование, но и резко снижает общее количество перетасовываемых записей. В случае задачи Златовласки, где в 25 % строк содержатся нулевые значения, в результате предыдущей итерации нашего решения остается отсортировать примерно 75 % кортежей.

[1] Использование распределенной операции reduceByKey позволяет понизить его просто до числа столбцов ценой перетасовки.

Агрегирование к ((значение ячейки, индекс столбца), количество) по каждой секции

Пример 6.21 содержит код для этого первого шага. Вместо отображения в виде пар (`значение ячейки, индекс столбца`) мы отображаем ((`значение ячейки, индекс столбца`), `количество`) по каждой из секций. Преобразуем столбцовые данные в пары с помощью операции `mapPartitions`. Теперь вместо выполнения операции `flatMap` над каждой строкой исходных данных обновляем объект типа `HashMap`, ключи которого представляют собой пары (`значение, индекс`), а значения — количества вхождений этой пары (`значение ячейки, индекс столбца`) в данной секции. Даже если все значения в каждом из столбцов заданной секции уникальны, данное решение не должно вызвать ошибок нехватки памяти. Мы создаем `HashMap` для каждой из секций, вследствие чего размеру `HashMap` нужно быть меньше итератора по всем записям. Однако это может оказаться не так, когда все значения действительно не повторяются, поскольку хеш-карта — намного менее эффективная структура данных, чем итератор.

Пример 6.21. Алгоритм решения задачи Златовласки. Версия 4, агрегирование по каждой из секций

```
def getAggregatedValueColumnPairs(dataFrame: DataFrame):
    RDD[((Double, Int), Long)] = {

  val aggregatedValueColumnRDD = dataFrame.rdd.mapPartitions(rows => {
    val valueColumnMap = new mutable.HashMap[(Double, Int), Long]()
    rows.foreach(row => {
      row.toSeq.zipWithIndex.foreach{ case (value, columnIndex) =>
        val key = (value.toString.toDouble, columnIndex)
        val count = valueColumnMap.getOrElseUpdate(key, 0)
        valueColumnMap.update(key, count + 1)
      }
    })

    valueColumnMap.toIterator
  })

  aggregatedValueColumnRDD
}
```

Сортировка и нахождение ранговых статистик

Оставшаяся часть функции аналогична исходной версии. Мы просто меняем код с целью отслеживать количество вхождений пары вместо допущения, что каждая пара в отсортированном наборе RDD встречается лишь один раз. Измененный код показан в примерах 6.22, 6.23, 6.24 и 6.25, а пример 6.26 объединяет все решение.

Пример 6.22. Алгоритм решения задачи Златовласки. Версия 4, подсчет значений по столбцам в каждой из секций

```
private def getColumnsFreqPerPartition(
  sortedAggregatedValueColumnPairs: RDD[((Double, Int), Long)],
  numOfColumns : Int): Array[(Int, Array[Long])] = {

  val zero = Array.fill[Long](numOfColumns)(0)

  def aggregateColumnFrequencies(
    partitionIndex : Int, pairs : Iterator[((Double, Int), Long)]) = {
    val columnsFreq : Array[Long] = pairs.aggregate(zero)(
       (a : Array[Long], v : ((Double, Int), Long)) => {
        val ((value, colIndex), count) = v
        a(colIndex) = a(colIndex) + count
        a},
       (a : Array[Long], b : Array[Long]) => {
        a.zip(b).map{ case(aVal, bVal) => aVal + bVal}
      })

    Iterator((partitionIndex, columnsFreq))
  }

  sortedAggregatedValueColumnPairs.mapPartitionsWithIndex(
    aggregateColumnFrequencies).collect()
}
```

Пример 6.23. Алгоритм решения задачи Златовласки. Версия 4, определение местоположения ранговых статистик в каждой из секций

```
private def getRanksLocationsWithinEachPart(targetRanks : List[Long],
    partitionColumnsFreq : Array[(Int, Array[Long])],
    numOfColumns : Int) : Array[(Int, List[(Int, Long)])] = {

  val runningTotal = Array.fill[Long](numOfColumns)(0)

  partitionColumnsFreq.sortBy(_._1).map { case (partitionIndex, columnsFreq)=>
    val relevantIndexList = new mutable.MutableList[(Int, Long)]()

    columnsFreq.zipWithIndex.foreach{ case (colCount, colIndex) =>
      val runningTotalCol = runningTotal(colIndex)

      val ranksHere: List[Long] = targetRanks.filter(rank =>
        runningTotalCol < rank && runningTotalCol + colCount >= rank)
      relevantIndexList ++= ranksHere.map(
        rank => (colIndex, rank - runningTotalCol))

      runningTotal(colIndex) += colCount
    }

    (partitionIndex, relevantIndexList.toList)
  }
}
```

Пример 6.24. Алгоритм решения задачи Златовласки. Версия 4, фильтрация ранговых статистик

```
private def findTargetRanksIteratively(
            sortedAggregatedValueColumnPairs : RDD[((Double, Int), Long)],
            ranksLocations : Array[(Int, List[(Int, Long)])]): RDD[(Int, Double)] =
{

  sortedAggregatedValueColumnPairs.mapPartitionsWithIndex((partitionIndex : Int,
    aggregatedValueColumnPairs : Iterator[((Double, Int), Long)]) => {

    val targetsInThisPart: List[(Int, Long)] = ranksLocations(partitionIndex)._2
    if (targetsInThisPart.nonEmpty) {
      FindTargetsSubRoutine.asIteratorToIteratorTransformation(
      aggregatedValueColumnPairs,
        targetsInThisPart)
    } else {
      Iterator.empty
    }
  })
}
```

Пример 6.25. Алгоритм решения задачи Златовласки. Версия 4, преобразование «итератор — итератор» для фильтрации ранговых статистик

```
def asIteratorToIteratorTransformation(
  valueColumnPairsIter : Iterator[((Double, Int), Long)],
  targetsInThisPart: List[(Int, Long)] ): Iterator[(Int, Double)] = {

  val columnsRelativeIndex = targetsInThisPart.groupBy(_._1).mapValues(_.map(_._2))
  val columnsInThisPart = targetsInThisPart.map(_._1).distinct

  val runningTotals : mutable.HashMap[Int, Long]= new mutable.HashMap()
    runningTotals ++= columnsInThisPart.map(columnIndex => (columnIndex, 0L)).toMap

  // Отфильтровываем пары, не содержащие индекса столбца из этой части
  val pairsWithRanksInThisPart = valueColumnPairsIter.filter{
    case (((value, colIndex), count)) =>
      columnsInThisPart contains colIndex
   }

  // Отображаем пары valueColumn в виде списка пар (colIndex, value),
  // соответствующих одной из требуемых ранговых статистик из этой секции
  pairsWithRanksInThisPart.flatMap{

    case (((value, colIndex), count)) =>

        val total = runningTotals(colIndex)
        val ranksPresent: List[Long] = columnsRelativeIndex(colIndex)
                                       .filter(index => (index <= count + total)
                                         && (index > total))

        val nextElems: Iterator[(Int, Double)] =
          ranksPresent.map(r => (colIndex, value)).toIterator

        // Обновляем значения промежуточных сумм
```

```
        runningTotals.update(colIndex, total + count)
        nextElems
  }
}
```

Объединяя весь код, получаем окончательное решение задачи Златовласки (см. пример 6.26).

Пример 6.26. Решение задачи Златовласки с помощью хеш-карты

```
def findRankStatistics(dataFrame: DataFrame, targetRanks: List[Long]):
  Map[Int, Iterable[Double]] = {

  val aggregatedValueColumnPairs: RDD[((Double, Int), Long)] =
    getAggregatedValueColumnPairs(dataFrame)
  val sortedAggregatedValueColumnPairs = aggregatedValueColumnPairs.sortByKey()
  sortedAggregatedValueColumnPairs.persist(StorageLevel.MEMORY_AND_DISK)

  val numOfColumns = dataFrame.schema.length
  val partitionColumnsFreq =
    getColumnsFreqPerPartition(sortedAggregatedValueColumnPairs, numOfColumns)
  val ranksLocations =
    getRanksLocationsWithinEachPart(targetRanks,
      partitionColumnsFreq, numOfColumns)

  val targetRanksValues =
    findTargetRanksIteratively(sortedAggregatedValueColumnPairs, ranksLocations)
  targetRanksValues.groupByKey().collectAsMap()
}
```

Как и ожидалось, это решение приводит к 4–5-кратному ускорению работы на не занятом другими задачами кластере.

Задача Златовласки: «разбор полетов»

Перед нами была поставлена задача найти произвольное количество ранговых статистик по группам (столбцам). Мы предложили пять ее решений.

- «Златовласка. Версия 0: итеративное решение» на с. 152. Наше первое решение основывалось на итеративном прохождении в цикле по каждой из групп и выполнении распределенной сортировки, что означало один этап и одну дорогостоящую распределенную сортировку для каждой из групп.
- «Златовласка. Версия 1: решение с использованием функции groupByKey» на с. 157. Данное решение задействует `groupByKey` для перемещения (путем перетасовки) относящихся к одной группе записей в одну секцию. После этого можно отсортировать каждую группу за один этап, прибегнув к функции `mapPartitions`.
- «Златовласка. Версия 2: вторичная сортировка» на с. 181. С помощью методики вторичной сортировки мы усовершенствовали основанное на функции

`groupByKey` решение, заменив операцию `groupByKey` и сортировку на функцию `repartitionAndSortWithinPartitions`, делегировав работу по сортировке каждой из групп этапу перетасовки.

- «Златовласка. Версия 3: сортировка по значениям ячеек» на с. 190. Далее мы поняли, что можно решить нашу задачу, обратившись вместо сортировки группы лишь к одной сортировке значений всех записей. Мы разработали решение, которое при значениях записей (а не индексах групп/столбцов) в качестве ключей сортирует все записи, после чего выполняет серию узких преобразований для сбора результатов. Ожидается, что новые ключи сортировки (значения в столбцах) будут содержать меньше дубликатов, чем размеры групп, использовавшиеся в качестве ключей в версии 2.
- «Златовласка. Версия 4: свертка до уникальных значений по каждой секции» на с. 193. И наконец, осознав наличие высокого процента дублирующихся значений в каждой из групп, мы модифицировали предыдущее решение и выполняем теперь свертку с отображением перед сортировкой данных. Это решение показывает лучшие результаты на предоставленных заказчиком асимметричных данных.

Мы выяснили, что производительность зависит в основном от трех характеристик входных данных: 1) количества исходных записей; 2) количества групп (столбцов, в данном случае), на основе которых вычисляются показатели; 3) доли дублирующихся записей из расчета на группу. Обратите внимание: все эти характеристики сильно зависят от размера и конфигурации используемого аппаратного обеспечения. При небольшом количестве записей они без проблем поместятся в памяти любого из исполнителей.

«Златовласка. Версия 4: свертка до уникальных значений по каждой секции» на с. 193 не лучшее решение для произвольных входных данных. А единственное решение из пяти, которое всегда работает «плохо», — применяющее операцию `groupByKey` (версия 1). Можно ожидать, что производительность версии 1 будет ниже, чем у решения с использованием `repartitionAndSortWithinPartitions` (версия 2) во всех случаях вследствие ограничений `groupByKey` и преимуществ вторичной сортировки (см. раздел «Вторичные сортировки и функция repartitionAndSortWithinPartitions» на с. 176). Каждая из оставшихся четырех версий может оказаться предпочтительной в определенных случаях.

Версия 0 сортирует значения в каждой из групп (как и решения 3 и 4). Это оптимально в случае одной-единственной группы, поскольку не требует лишних проходов по данным.

Версия 2 секционирует по группам. Это оптимально в случае многочисленных групп, каждая из которых легко умещается в оперативной памяти отдельного исполнителя. Если эти группы относительно невелики, то производительность данного решения окажется выше, чем у версий 3 и 4, сортирующих по значению и требующих трех узких преобразований для вычисления правильных результатов, поскольку нуждается в меньшем количестве проходов по данным. При достаточном

количестве групп и ранговых статистик это решение скорее отработает успешно, чем версии 3 и 4, которые обе требуют хранения в оперативной памяти хеш-карты ранговых статистик для каждой из групп.

Версия 3 секционирует по значениям по всем группам. Она лучше всего подходит в случае, когда все группы очень велики, а в столбцах мало дублирующихся значений. В отличие от версии 2 только записи с дублирующимися значениями обязаны располагаться в одной секции. Это приводит к лучшему распараллеливанию при условии, что количество дублирующихся значений меньше максимального размера группы.

Версия 4 использует узкое преобразование для агрегирования уникальных значений по группам в исходных секциях с последующим секционированием по уникальным значениям (аналогично решению 3). Оно лучше решения 3 только в случае большого количества дублирующихся значений в столбцах. Однако если все значения уникальны, то данная операция никаких выгод не дает, а только ведет к риску ошибок нехватки памяти из-за создания хеш-карт всех значений в каждой из секций.

Наш сценарий использования включал входные данные, содержавшие несколько тысяч групп по 300 миллионов записей каждая. В каждом столбце (группе), как мы видели, около 25 % записей повторялись. В десятиузловом загруженном кластере версии 0, 1 и 2 вообще не отрабатывали. Версия 3 отработала, но по сравнению с ней версия 4 показала четырехкратный рост производительности.

Резюме

В данной главе мы изучили применение функций из классов `PairRDDFunctions` и `OrderedRDDFunctions` наиболее рациональными для использования в больших масштабах способами. Большая часть главы была посвящена специализированным методикам работы с широкими преобразованиями. Мы сосредоточились на случаях ошибок нехватки памяти во время этапа перетасовки, в частности на том, что следует избегать использования таких операций, как `groupByKey`, не снижающих объем памяти, необходимой для хранения всех относящихся к каждому ключу значений. Мы разобрались с секционированием: научились продумывать наперед следующее преобразование пар «ключ — значение» и выполнять интеллектуальное секционирование, снижающее количество необходимых перетасовок. Мы изучили некоторые стратегии понижения количества перетасовок: использование интеллектуального секционирования, сохранение информации о секционировании при узких преобразованиях, а также применение совместного расположения при соединениях. Мы рассмотрели отдельные методики снижения стоимости перетасовок в тех случаях, когда они необходимы. В частности, показали, что несбалансированные данные, особенно с большим количеством дублирующихся значений на один ключ, с большой вероятностью приводят к замедлению работы перетасовок и вызывают ошибки памяти.

Надеемся, что, помимо уловок, позволяющих повысить производительность преобразований наборов RDD, в этой главе вы обнаружили и некоторые средства переосмысления концепций решения задач с помощью фреймворка Spark. Сосредоточившись на случае задачи Златовласки, мы попытались показать, что написание производительного кода Spark иногда требует нестандартного мышления. Простое соединение набора вызовов API, описывающих простейшим образом желаемые действия, чаще всего приводит к решению наподобие представленного в разделе «Златовласка. Версия 1: решение с использованием функции groupByKey» на с. 157: реализация, которая нормально работает только на относительно маленьких, чистых входных данных и вообще не работает на большинстве реальных наборов данных. Мы попытались показать, что, несмотря на относительную простоту написания распределенного кода с помощью фреймворка Spark, оптимальных результатов можно добиться, только уделив должное внимание выполнению этих распределенных вычислений. Мы надеемся, что убедили вас думать о ключах в Spark не как об индексе или категории для записей, а как о центральном элементе распараллеливания вычислительной нагрузки. Призываем вас при любой возможности проектировать программы Spark с максимальным учетом имеющейся информации об объеме, распределении и сложности обрабатываемых ими данных. С философской точки зрения мы хотели продемонстрировать, что высокопроизводительный код зачастую требует больших, а не маленьких изменений — и наиболее эффективный код далеко не всегда самый «чистый».

7 Выходим за рамки Scala

При работе с фреймворком Spark вы не обязаны ограничиваться одним только языком Scala либо языками, которые явно поддерживает Spark или даже JVM. Apache предусмотрел в Spark несколько API для написания программ драйверов и кода работников на языках R[1], Python, Scala и Java, а также сторонние привязки[2] для дополнительных языков, включая JavaScript, Julia, C# и F#. Совместимость Spark с языками можно рассматривать на двух уровнях: первый — рабочий код внутри преобразований (например, лямбда-выражения при отображениях), а второй — код для определения преобразований над наборами RDD/`Dataset` (например, драйверная программа). В этой главе мы обсудим вопросы производительности при использовании других языков программирования в Spark, а также способы эффективной работы с существующими библиотеками.

Зачастую для написания кода внутри преобразований выбирается тот же язык программирования, что и для написания драйверной программы, но при работе со специализированными библиотеками или инструментами (например, CUDA[3]) написать всю программу на одном языке бывает непросто, если вообще возможно. Фреймворк Spark поддерживает использование на драйвере множества языков, и еще большее количество языков можно задействовать в преобразованиях на рабочих узлах. Хотя API для разных языков схожи, характеристики производительности существенно различаются, если код выполняется вне JVM. Мы обсудим

---

1 Существует несколько конкурирующих API для языка R, но из соображений производительности их внутренняя архитектура одинакова.

2 Поддержка их производителем не означает быстроты работы; в некоторых случаях в сторонних библиотеках привязки для других языков была проведена значительная работа по минимизации накладных расходов, чего в основных языках реализовано не было.

3 CUDA — специализированный язык для выполнения параллельных вычислений на графических картах от компании NVIDIA.

архитектуру поддержки различных языков, а также влияние различий в производительности на вашу работу.

В целом библиотеки, осуществляющие привязку к не-JVM-языкам программирования, обращаются к Java-интерфейсу Spark с помощью механизма удаленных вызовов (например, Py4J), передавая сериализованное представление кода, который нужно выполнить на рабочем узле. Вне зависимости от языка программирования, используемого для драйверной программы, работники Spark выполняют код в виртуальной машине Java и при необходимости вызывают ориентированную на конкретный язык исполняющую программу. Если для нужного языка программирования отсутствует библиотека привязки для драйвера Spark, то помните, что можно написать собственные преобразования для обращения к программам на другом языке программирования.

На своей стороне работник Spark всегда действует в JVM и в случае надобности запускает дополнительный целевой процесс, копируя туда требуемые данные и результат. Это копирование является дорогостоящим, но DAG зависимостей Spark и искусное построение конвейера минимизирует количество случаев, когда оно необходимо. Методики стыковки с кодом, задействуемые API других языков, аналогичны методикам, которые применяются для обращения к пользовательскому коду, независимо от языка драйвера.

Существует множество способов выйти за рамки JVM, от Java Native Interface (JNI) и конвейеров Unix до взаимодействия с работающими на протяжении длительного времени вспомогательными серверами через сокеты. Эти же методики используются внутри Spark для стыковки с другими языками программирования. Например, JNI применяется для обращения к некоторым библиотекам линейной алгебры, а конвейеры Unix — для обеспечения взаимодействия с кодом на языке Python. Оптимальное решение часто зависит от того, нужно ли совершать несколько преобразований, от затрат на настройку среды выполнения и языка программирования, а также от вычислительной сложности преобразований. Независимо от конкретного подхода к интеграции языков, не использующих JVM, все они в настоящее время требуют копирования данных из нее в среду выполнения целевого языка. Проводимые сейчас работы по интеграции как Tungsten, так и Arrow позволяют надеяться, что в будущем работать с данными из Spark за пределами JVM станет проще.

Не все языки программирования требуют выхода за рамки JVM, и их применение со Spark позволяет избежать дорогостоящего копирования данных из работника Spark в программу на требуемом языке. Отдельные языки программирования используют смешанный подход, например проект Eclair JS (см. подраздел «Как работает Eclair JS» на с. 218), в котором экземпляр Spark запускается в JVM, но драйверная программа работает вне виртуральной машины. Конечно, несмотря на дополнительные затраты на работу драйверной программы вне JVM, объемы передаваемых между драйвером Scala и целевым драйвером данных намного меньше объемов данных, обрабатываемых одним-единственным рабочим узлом.

За пределами Scala, но в рамках JVM

В этом разделе мы рассмотрим доступ к API Spark из различных языков программирования в JVM, а также некоторые вопросы производительности при выходе за пределы языка Scala. Даже если вы работаете вне JVM, данный раздел может оказаться полезен, поскольку не-JVM-языки часто зависят от API Java, а не от API Scala.

Работа на других языках программирования далеко не всегда означает необходимость выхода за пределы JVM, и работа в JVM имеет немало преимуществ с точки зрения производительности — в основном вследствие того, что не требуется копировать данные. Хотя для обращения к Spark не из языка Scala не обязательно нужны специальные библиотеки привязки или адаптеры, вызвать код на языке Scala из других языков программирования может быть непросто. Фреймворк Spark поддерживает использование в преобразованиях лямбда-выражений языка Java 8, а у тех, кто применяет более старые версии JDK, есть возможность реализовать соответствующий интерфейс из пакета `org.apache.spark.api.java.function`. Даже в случаях, когда не требуется копировать данные, у работы на другом языке программирования могут быть небольшие, но важные нюансы, связанные с производительностью.

Особенно ярко сложности с обращением к различным API Scala проявляют себя при вызове функций с тегами классов или при использовании свойств, предоставляемых с помощью неявных преобразований типов (например, всей относящейся к классам `Double` и `Tuple` функциональности наборов RDD). Для механизмов, зависящих от неявных преобразований типов, часто предоставляются эквивалентные конкретные классы наряду с явными преобразованиями к ним. Функциям, зависящим от тегов классов, можно передавать фиктивные теги классов (скажем, `AnyRef`), причем зачастую адаптеры делают это автоматически. Применение конкретных классов вместо неявного преобразования типов обычно не приводит к дополнительным накладным расходам, но фиктивные теги классов могут накладывать ограничения на некоторые оптимизации компилятора.

API Java не слишком отличается от API Scala в смысле свойств, лишь изредка отсутствуют некоторые функциональные возможности или API разработчика. Поддержка других языков программирования JVM, например языка Clojure с DSL Flambo (https://github.com/yieldbot/flambo) и библиотеки sparkling (https://github.com/gorillalabs/sparkling/), осуществляется с помощью различных API Java вместо непосредственного вызова API Scala. Поскольку большинство привязок языков, даже таких не-JVM-языков, как Python и R, идет через API Java (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.api.java.package), то полезно будет разобраться с ним.

API Java очень напоминают API Scala, хотя и не зависят от тегов классов и неявных преобразований. Отсутствие последних означает, что вместо автоматического преобразования наборов RDD объектов `Tuple` или `double` в специальные классы с дополнительными функциями приходится использовать функции явного преобразования типа (например, `mapToDouble` или `mapToPair`). Указанные функции определены только для наборов RDD языка Java; к счастью для совместимости, эти

специальные типы представляют собой просто адаптеры для наборов RDD языка Scala. Кроме того, эти специальные функции возвращают различные типы данных, такие как `JavaDoubleRDD` и `JavaPairRDD`, с возможностями, предоставляемыми неявными преобразованиями языка Scala.

Вновь обратимся к каноническому образцу подсчета слов, воспользовавшись API Java (пример 7.1). Поскольку вызов API Scala из Java может иногда оказаться непростым делом, то API Java фреймворка Spark почти все реализованы на языке Scala со спрятанными тегами классов и неявными преобразованиями. Благодаря этому адаптеры Java представляют собой очень тонкий слой, в среднем состоящий лишь из нескольких строк кода, и их переписывание практически не требует усилий.

Пример 7.1. Подсчет слов (Java)

```
import scala.Tuple2;

import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;

import java.util.regex.Pattern;
import java.util.Arrays;

public final class WordCount {
  private static final Pattern pattern = Pattern.compile(" ");

  public static void main(String[] args) throws Exception {
    JavaSparkContext jsc = new JavaSparkContext();
    JavaRDD<String> lines = jsc.textFile(args[0]);
    JavaRDD<String> words = lines.flatMap(e -> Arrays.asList(
                                        pattern.split(e)).iterator());
    JavaPairRDD<String, Integer> wordsIntial = words.mapToPair(
      e -> new Tuple2<String, Integer>(e, 1));
  }
}
```

Фреймворк Spark поддерживает лямбда-выражения Java 8 для большинства преобразований. Если вы работаете с более ранней версией Java, то придется создать экземпляры интерфейсов из пакета org.apache.spark.api.java.function.package (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.api.java.function.package). Имена функций обычно схожи с названием преобразования (например, FlatMapFunction или DoubleFunction).

Иногда может понадобиться преобразовать наборы RDD Java в наборы RDD Scala или наоборот. Чаще всего это бывает нужно для библиотек, требующих на входе или возвращающих наборы RDD Scala, но иногда базовые свойства Spark могут еще не быть доступны в API Java. Преобразование набора RDD Java в набор RDD Scala — простейший способ использовать эти новые свойства.

При необходимости передать набор RDD Java в библиотеку Scala, ожидающую на входе обычный RDD Spark, получить доступ к находящемуся в его основе RDD Scala можно с помощью метода `rdd()`. Чаще всего этого оказывается достаточно для передачи итогового RDD в любую нужную библиотеку Scala; в числе заслуживающих упоминания исключений — библиотеки Scala, полагающиеся в своей работе на неявные преобразования типов содержимого наборов или информацию тегов классов. В таком случае простейшим способом обращения к неявным преобразованиям будет написание небольшого адаптера на Scala. Если Scala-оболочки использовать нельзя, то можно вызвать соответствующую функцию класса JavaConverters (http://www.scala-lang.org/api/2.12.0/scala/collection/JavaConverters$.html) и сформировать фиктивный тег класса.

Для создания фиктивного тега класса можно использовать метод `scala.reflect.ClassTag$.MODULE$.AnyRef()` или получить настоящий с помощью `scala.reflect.ClassTag$.MODULE$.apply(CLASS)`, как показано в примерах 7.2 и 7.3.

Для преобразования из RDD Scala в RDD Java информация о теге класса часто важнее, чем для большинства библиотек Spark. Причина в том, что, хотя различные классы `JavaRDD` предоставляют общедоступные конструкторы, принимающие RDD Scala в качестве аргументов, они предназначены для вызова из кода на языке Scala, а потому требуют информации о теге класса.

Если речь идет о проекте или библиотеке, использующих несколько языков программирования, то стоит обдумать возможность создания RDD Java в коде на Scala, где проще получить доступ к информации о теге класса.

Фиктивные теги классов чаще всего используются в обобщенном или шаблонизированном коде, где точные типы неизвестны в момент компиляции. Таких тегов часто бывает достаточно, хотя существует возможность потери некоторых нюансов на стороне Scala-кода; в очень редких случаях для кода на языке Scala необходима точная информация о теге класса. В этом случае *придется* использовать настоящий тег. В большинстве случаев это требует не намного больших усилий и улучшает производительность, так что старайтесь использовать такие теги везде, где только возможно.

Пример 7.2. Обеспечение совместимости RDD Java/Scala с помощью фиктивного тега класса

```
public static JavaPairRDD wrapPairRDDFakeCt(
  RDD<Tuple2<String, Object>> RDD) {
  // Формируем теги классов путем приведения типа AnyRef — это чаще
  // всего делается в случае обобщенного или шаблонизированного кода,
  // когда невозможно явным образом сформировать правильный тег класса,
  // поскольку использование фиктивного тега класса может привести
  // к снижению производительности
  ClassTag<Object> fake = ClassTag$.MODULE$.AnyRef();
  return new JavaPairRDD(rdd, fake, fake);
}
```

Пример 7.3. Обеспечение совместимости RDD Java/Scala

```
public static JavaPairRDD wrapPairRDD(
  RDD<Tuple2<String, Object>> RDD) {
  // Формируем теги классов
  ClassTag<String> strCt = ClassTag$.MODULE$.apply(String.class);
  ClassTag<Long> longCt = ClassTag$.MODULE$.apply(scala.Long.class);
  return new JavaPairRDD(rdd, strCt, longCt);
}
```

API как Spark SQL, так и конвейера ML были по большей части сделаны единообразно в Java и Scala. Однако существуют предназначенные для Java вспомогательные функции, а функции языка Scala, эквивалентные им, вызвать непросто. Вот их примеры: различные числовые функции, такие как `plus`, `minus` и т. д., для класса `Column`. Вызвать их перегруженные эквиваленты из языка Scala (`+`, `-`) сложно. Вместо использования `JavaDataFrame` и `JavaSQLContext` необходимые для Java методы сделаны доступными в `SQLContext` и обычных наборах `DataFrame`. Это может смутить вас, ведь некоторые упомянутые в документации по Java методы нельзя задействовать из кода на языке Java, но в подобных случаях для вызова из Java предоставляются функции с аналогичными названиями.

Пользовательские функции (UDF) в языке Java, а если уж на то пошло, и в большинстве других языков, кроме Scala, требуют указания типа возвращаемого функцией значения, поскольку его невозможно логически вывести, подобно тому как это выполняется в языке Scala (пример 7.4).

Пример 7.4. Образец UDF для языка Java

```
sqlContext.udf()
  .register("strlen",
            (String s) -> s.length(), DataTypes.StringType);
```

Хотя необходимые для API Scala и Java типы различаются, обертывание типов-коллекций языка Java не требует дополнительного копирования. В случае итераторов требуемое для адаптера преобразование типа выполняется отложенным образом по мере обращения к элементам, что позволяет фреймворку Spark сбрасывать данные в случае надобности (как обсуждалось в разделе «Выполнение преобразований “итератор — итератор” с помощью функции mapPartitions» на с. 121). Это очень важно, поскольку для многих простых операций стоимость копирования данных может оказаться выше затрат на само вычисление.

В ранних версиях фреймворка Spark для API Java ошибочно требовался объект типа Iterable вместо Iterator, что ограничивало возможности создания преобразований «итератор — итератор» в коде на языке Java.

За пределами и Scala, и JVM

Если не ограничивать себя JVM, то количество доступных для работы языков программирования резко возрастает. Однако при текущей архитектуре Spark работа вне JVM — особенно на рабочих узлах — может приводить к существенному росту затрат из-за копирования данных в рабочих узлах между JVM и кодом на целевом языке. При сложных операциях доля затрат на копирование данных относительно невелика, но при простых она легко способна привести к удвоению общих вычислительных затрат.

Первый непосредственно поддерживаемый вне Spark не-JVM-язык программирования — Python, его API и интерфейс стали образцом, на котором основываются реализации для остальных не-JVM-языков программирования.

Как работает PySpark

PySpark подключается к JVM Spark с помощью смеси каналов на работниках и Py4J — специализированной библиотеки, обеспечивающей взаимодействие Python/Java — на драйвере. Под этой, на первый взгляд, простой архитектурой скрывается немало сложных нюансов, благодаря которым работает PySpark, как показано на рис. 7.1. Одна из основных проблем: даже когда данные скопированы из работника Python в JVM, они находятся не в том виде, который может легко разобрать виртуальная машина. Необходимы специальные усилия на стороне и работника Python, и Java, чтобы гарантировать наличие в JVM достаточного объема информации для таких операций, как секционирование.

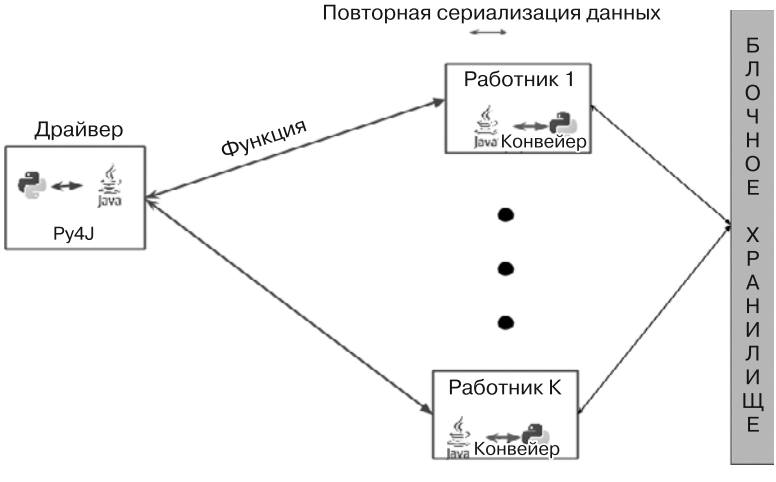

Рис. 7.1. Архитектура PySpark

Необходимо передать данные по рабочим узлам между JVM и Python после первоначального чтения данных из устойчивого хранилища (например, HDFs или S3) и до первой перетасовки.

У вас проблема с IPython? В Spark 2.0+ старый синтаксис для открытия блокнота IPython поменялся с IPYTHON_OPTS="notebook" на PYSPARK_DRIVER_ PYTHON="ipython" PYSPARK_DRIVER_ PYTHON_OPTS="notebook".

Наборы RDD PySpark

Затраты ресурсов на передачу данных в JVM и из нее, а также на запуск исполнителя Python весьма значительны. Избежать многих проблем с производительностью с API наборов RDD PySpark можно, используя API `DataFrame`/`Dataset`, благодаря тому что данные при этом как можно дольше остаются в JVM.

Копирование данных из JVM в Python выполняется с помощью сокетов и сериализованных байтов. Более общая версия для взаимодействия с программами на других языках доступна через интерфейс `PipedRDD`, применение которого показано в подразделе «Использование pipe» на с. 220.

Организация каналов для обмена данными (в двух направлениях) для каждого преобразования была бы слишком дорогостоящей. Вследствие этого PySpark организует (при возможности) конвейер преобразований Python внутри интерпретатора Python, соединяя в цепочку операцию `filter`, а после нее — `map`, на итераторе Python-объектов с помощью специализированного класса `PipelinedRDD`. Даже когда нужно перетасовать данные и PySpark не способен связать преобразования цепочкой в виртуальной машине отдельного работника, можно повторно использовать интерпретатор Python, так что затраты на запуск интерпретатора не приведут к дальнейшему замедлению работы.

Это только часть головоломки. Обычные `PipedRDD` работают с типом `String`, перетасовывать который не так уж просто из-за отсутствия естественного ключа. В PySpark же, а по его образу и подобию в библиотеках привязки ко многим другим языкам программирования, применяется специальный тип `PairwiseRDD`, где ключ представляет собой длинное целое, а его десериализация выполняется пользовательским кодом на языке Scala, предназначенном для синтаксического разбора Python-значений. Затраты на эту десериализацию не слишком велики, но она демонстрирует, что Scala в фреймворке Spark в основном рассматривает результаты работы кода Python как «непрозрачные» байтовые массивы.

Поскольку сериализация и десериализация требуют некоторых затрат вычислительных ресурсов, то в PySpark используется пакетный сериализатор, что иногда может приводить к неожиданным последствиям (например, при повторном секционировании PySpark не разделяет между собой элементы, относящиеся к одному пакету).

При всей его простоте этот подход к интеграции работает на удивление хорошо, причем в языке Python доступно большинство операций над наборами RDD Scala. В некоторых наиболее сложных местах кода происходит обращение к библиотекам, например MLlib, а также загрузка/сохранение данных из различных источников.

Работа с различными форматами данных тоже накладывает свои ограничения, поскольку значительная часть кода загрузки/сохранения данных фреймворка Spark основана на Java-интерфейсах Hadoop. Это значит, что все загружаемые данные сначала загружаются в JVM, а лишь потом перемещаются в Python.

Для взаимодействия с MLlib обычно применяются два подхода: или в PySpark используется специализированный тип данных с преобразованиями типов Scala, или алгоритм заново реализуется в Python. Этих проблем можно избежать с помощью пакета Spark ML, в котором применяется интерфейс `DataFrame`/`Dataset`, обычно хранящий данные в JVM.

Наборы DataFrame и Dataset пакета PySpark

Наборы `DataFrame` и `Dataset` лишены многих проблем с производительностью API наборов RDD Python благодаря тому, что хранят данные в JVM как можно дольше. Тот же тест производительности, который мы провели для иллюстрации превосходства наборов `DataFrame` над наборами RDD (см. рис. 3.1), показывает значительные различия при запуске в Python (рис. 7.2).

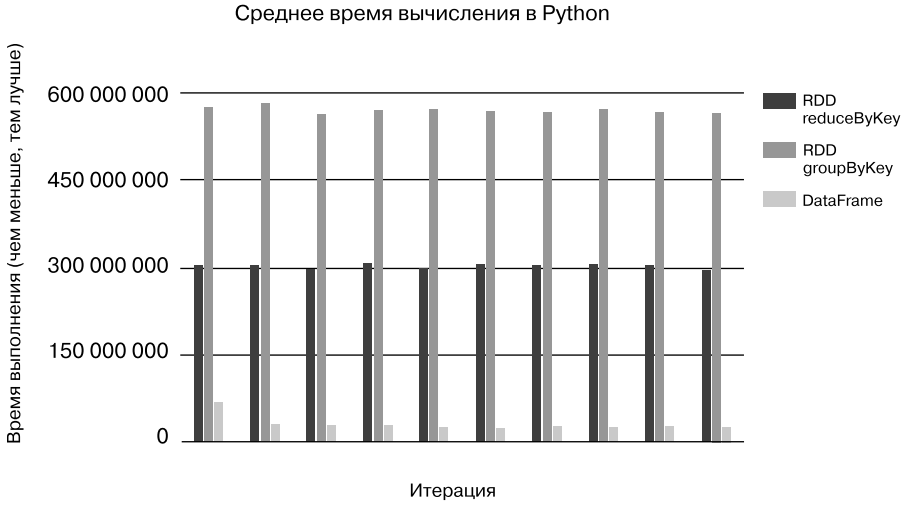

Рис. 7.2. Производительность Spark SQL в Python

При многих операциях с наборами `DataFrame` и `Dataset`, возможно, вообще не потребуется перемещать данные из JVM, хотя использование различных UDF, UDAF и лямбда-выражений языка Python, естественно, требует перемещения

части данных в JVM. Это приводит к следующей упрощенной схеме для многих операций, выглядящей так, как показано на рис. 7.3.

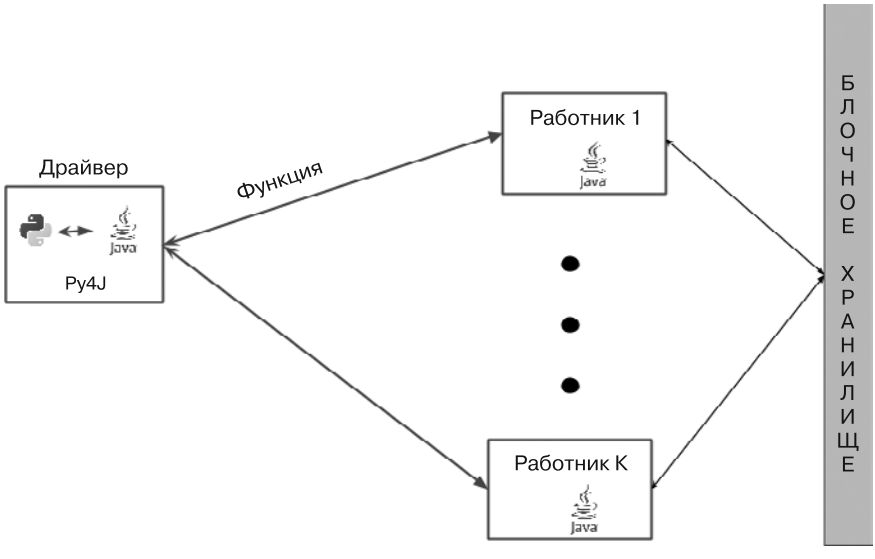

Рис. 7.3. Схема выполнения PySpark SQL

PySpark не задействует проект Jython, поскольку оказалось, что многим пользователям требуется доступ к таким библиотекам, как numpy, scipy и pandas, плохо работающим в Jython.

Были начаты предварительные исследования, касающиеся возможности применять Jython для ускорения работы пользовательских функций Python, в которых не используются расширения языка C. Следить за продвижением этих работ можно в SPARK-15369 (https://issues.apache.org/jira/browse/SPARK-15369).

Доступ к нижележащим Java-объектам и смешанному коду на Scala

Важное следствие архитектуры PySpark состоит в том, что многие из Python-классов фреймворка Spark фактически являются адаптерами, служащими для трансляции вызовов из кода на Python в понятную JVM-форму.

Если вы сотрудничаете с разработчиками на Scala/Java и хотите взаимодействовать с их кодом, то заранее никаких адаптеров для обращения к вашему коду

не будет, но вы можете зарегистрировать свои Java/Scala UDF и воспользоваться ими из кода на Python. Начиная со Spark 2.1, это можно сделать с помощью метода `registerJavaFunction` объекта `sqlContext`.

Иногда эти адаптеры не имеют всех необходимых механизмов, и, поскольку в языке Python отсутствует жесткая защита от обращения к приватным методам, можно сразу обратиться к JVM. Такая же методика позволит обратиться к собственному коду в JVM и с небольшими усилиями преобразовать результаты обратно в объекты Python.

Хотя API библиотеки Py4J доступен, эти методики зависят от нюансов реализации PySpark, которые могут меняться от версии к версии.

В подразделе «Большие планы запросов и итеративные алгоритмы» на с. 91 мы отмечали важность использования JVM-версии наборов `DataFrame` и RDD в целях сокращения плана запроса. Это обходной путь, ведь когда планы запросов становятся слишком большими для обработки оптимизатором Spark SQL, SQL-оптимизатор, из-за помещения набора RDD в середину теряет возможность заглянуть за пределы момента появления данных в RDD. Того же можно добиться с помощью общедоступных API Python, однако при этом потеряются многие преимущества наборов `DataFrame`, ведь все данные должны будут пройти туда и обратно через рабочие узлы Python. Вместо этого можно сократить граф происхождения, продолжая хранить данные в JVM (как показано в примере 7.5).

Пример 7.5. Усечение большого плана запроса для набора DataFrame с помощью Python

```
def cutLineage(df):
    """
    Усечение графа происхождения DataFrame – используется для итеративных алгоритмов

    .. Примечание: эта функция использует внутренние члены классов
    и может перестать работать в следующих версиях
    >>> df = RDD.toDF()
    >>> cutDf = cutLineage(df)
    >>> cutDf.count()
    3
    """
    jRDD = df._jdf.toJavaRDD()
    jSchema = df._jdf.schema()
    jRDD.cache()
    sqlCtx = df.sql_ctx
    try:
        javaSqlCtx = sqlCtx._jsqlContext
    except:
        javaSqlCtx = sqlCtx._ssql_ctx
    newJavaDF = javaSqlCtx.createDataFrame(jRDD, jSchema)
    newDF = DataFrame(newJavaDF, sqlCtx)
    return newDF
```

Вообще говоря, по соглашению для доступа к внутренним Java-версиям большинства объектов Python используется синтаксис `_j[сокращенное_наименование]`. Так, например, у объекта `SparkContext` есть `_jsc`, который позволяет получить внутренний Java-объект `SparkContext`. Это возможно только в драйверной программе, так что при отправке PySpark-объектов на рабочие узлы вы не сможете получить доступ к внутреннему Java-компоненту и большая часть API работать не будет.

API языка Python обычно создают адаптеры для Java-версий API, а не сами Scala-версии.

Для обращения к классу Spark в JVM, у которого нет Python-адаптера, можно воспользоваться шлюзом Py4J на драйвере. Объект `SparkContext` содержит ссылку на шлюз в свойстве `_gateway`. Обратиться к любому Java-объекту позволит синтаксис `sc._gateway.jvm.[полное_имя_класса_в_JVM]`.

Py4J весьма зависит от рефлексии при определении того, какие методы вызывать. Обычно это не вызывает проблем, но способно повлечь сложности при использовании числовых типов. Попытка передать функции Scala при вызове параметра типа Integer, в то время как она ожидает тип Long, приведет к сообщению о невозможности найти метод, хотя в языке Python данное различие обычно не имеет значения.

Подобная методика сработает и для ваших собственных классов Scala, если они располагаются в соответствии с путем к классам. Добавить файлы JAR в путь к классам можно с помощью команды `spark-submit` с параметром `--jars` или задав свойства конфигурации `spark.driver.extraClassPath`. Пример 7.6, который помог сгенерировать рис. 7.2, умышленно устроен так, что генерирует данные для тестирования производительности с помощью существующего кода на языке Scala.

Пример 7.6. Вызов не-Spark-JVM-классов с помощью Py4J

```
sc = sqlCtx._sc
# Получение SQL Context, синтаксис версий 2.1, 2.0 и более ранних,
# чем 2.0, — ух ты, какие нюансы :p
try:
    try:
        javaSqlCtx = sqlCtx._jsqlContext
    except:
        javaSqlCtx = sqlCtx._ssql_ctx
except:
    javaSqlCtx = sqlCtx._jwrapped
jsc = sc._jsc
```

```
scalasc = jsc.sc()
gateway = sc._gateway
# Вызов java-метода, возвращающего набор RDD JVM-объектов
# Row (Int, Double). Хотя наборы RDD языка Python представляют собой
# обернутые RDD языка Java (даже наборы объектов Row), содержимое их
# различается, так что непосредственное его обертывание невозможно.
# Он возвращает Java-RDD объектов Row — обычно лучше было бы
# вернуть непосредственно набор DataFrame, но для целей иллюстрации
# мы воспользуемся набором RDD объектов Row.
java_rdd = (gateway.jvm.com.highperformancespark.examples.
            tools.GenerateScalingData.
            generateMiniScaleRows(scalasc, rows, numCols))
# Схемы сериализуются в формат JSON и пересылаются туда и обратно.
# Формируем Python-схему и преобразуем ее в Java-схему.
schema = StructType([
    StructField("zip", IntegerType()),
    StructField("fuzzyness", DoubleType())])
# Синтаксис версии 2.1 / до 2.1
try:
    jschema = javaSqlCtx.parseDataType(schema.json())
except:
    jschema = sqlCtx._jsparkSession.parseDataType(schema.json())
# Преобразуем RDD (Java) в DataFrame (Java)
java_dataframe = javaSqlCtx.createDataFrame(java_rdd, jschema)
# Обертываем DataFrame (Java) в DataFrame (Python)
python_dataframe = DataFrame(java_dataframe, sqlCtx)
# Преобразуем DataFrame (Python) в набор RDD
pairRDD = python_dataframe.rdd.map(lambda row: (row[0], row[1]))
return (python_dataframe, pairRDD)
```

Попытка использовать внутри преобразований Py4J в качестве связующего звена приведет к сбою во время выполнения.

Хотя многие классы языка Python представляют собой просто адаптеры Java-объектов, далеко не все Java-объекты можно обернуть в Python-объекты и затем использовать в Spark. Например, объекты в наборах RDD PySpark представлены в виде сериализованных строк, выполнить синтаксический разбор которых с легкостью можно только в коде на языке Python. К счастью, объекты `DataFrame` стандартизированы между разными языками программирования, так что если вы сумеете преобразовать свои данные в наборы `DataFrame`, то сможете затем обернуть их в Python-объекты и либо задействовать непосредственно в виде `DataFrame` языка Python, либо преобразовать `DataFrame` языка Python в RDD этого же языка.

Различные UDF и UDAF на языке Scala можно использовать из Python непосредственно, не прибегая к API Py4J.

Управление зависимостями PySpark

Зачастую основной причиной для использования отличного от Scala языка является желание задействовать существующие для этого языка библиотеки. Помимо ориентированных на конкретный язык библиотек, может понадобиться включить библиотеки самого Spark, особенно при работе с различными форматами данных. Существует несколько различных вариантов применения библиотек в PySpark, ориентированных как на Spark в целом, так и на конкретный язык.

Spark Packages (https://spark-packages.org/) представляет собой систему, позволяющую с легкостью включать JVM-зависимости в Spark. Дополнительные JVM-библиотеки в PySpark часто бывают нужны для поддержки дополнительных форматов данных.

При работе в командной оболочке Scala аргумент командной строки `--packages` позволит указать Maven-координаты требуемого пакета в оболочке. При сборке пакета Scala можно также вставить соответствующие требования в `.jar` сборки.

В Python можно создать Java- или Scala-проект с JVM-зависимостями и добавить в него `.jar` с помощью аргумента `--jar`. При работе в оболочке PySpark аргументы командной строки не допускаются, так что вместо этого можно задать значение переменной конфигурации `spark.jars.packages`.

При использовании Spark Packages зависимости автоматически подтягиваются из Maven и распространяются по кластеру. Если же нужные JVM-зависимости недоступны в Maven, то можно применить методику, аналогичную той, что приведена ниже для локальных записей Python.

Добавлять локальные зависимости с помощью PySpark можно как во время отправки задания, так и динамически, с помощью объекта `SparkContext`. В случае JVM-требований локальные зависимости могут быть файлами `.jar`, а в случае зависимостей Python — файлами `.zip` и `.egg`, автоматически добавляемыми в переменную `PYTHONPATH`.

В настоящее время ведется работа по разрешению программам Python Spark указывать необходимые пакеты pip и автоматически их устанавливать, но это предложение пока еще не принято. Его статус можно посмотреть в запросе на внесение изменений (https://github.com/apache/spark/pull/12398) и в Spark-5929 (https://issues.apache.org/jira/browse/SPARK-5929).

Те, кто работает с CDH-кластером, легко могут добавлять пакеты с помощью дистрибутива Anaconda. В сообщении из блога компании Cloudera под названием Making Python on Apache Hadoop Easier («Упрощаем работу с языком Python в кластерах Apache Hadoop») (http://blog.cloudera.com/blog/2016/02/making-python-on-apache-hadoop-easier-with-anaconda-and-cdh/) подробно рассказывается, как установить

пакеты в кластере. Чтобы эти пакеты стали доступны Apache Spark, достаточно установить значение переменной командной среды `PYSPARK_PYTHON` равным `/opt/cloudera/parcels/Anaconda/bin/python` в файле `spark-env.sh` или путем экспорта ее в профиле командной оболочки.

Если ни один из вышеописанных способов не работает при вашей конфигурации кластера, то остаются лишь несколько вариантов, каждый из которых неидеален. Простейший из них, весьма неуклюжий, состоит в импорте явным образом вашими преобразованиями нужного пакета, а в случае неудачной попытки импорта — выполнении установки с помощью pip. Аналогичный подход возможен с использованием транслирующих переменных или карты настроек при запуске программы. Если это не получится, то можно попросить администратора кластера установить пакет для всей системы, задействуя утилиту parallel-ssh или какую-либо аналогичную, как показано в примере 7.7.

Пример 7.7. Установка пакетов pip с помощью утилиты parallel-ssh

```
parallel-ssh pip install -h ./conf/slaves
```

Установка PySpark

Поддерживаемые Apache языки для фреймворка Spark не требуют отдельной установки, но, как упоминалось, у Python — свои механизмы управления пакетами.

Установка с помощью системы управления пакетами pip появилась в версии 2.1 Python, и в настоящий момент пакет PySpark можно скачать с сервера-зеркала загрузок Apache и выполнить команду `pip install pyspark-2.1.0.tar.gz`, обеспечивающую также поддержку VirtualEnv. Будущие версии PySpark, вероятно, станут непосредственно публиковаться в `PyPi`, благодаря чему будет возможен еще более простой синтаксис: `pip install pyspark`. После установки PySpark с помощью pip вы можете запустить свой любимый интерпретатор языка Python и импортировать `pyspark`, подобно любому другому пакету, или запустить командную оболочку PySpark командой `pyspark`.

Важно отметить, что такая установка с помощью pip не обязательна. При желании можно просто запустить PySpark из обычного установочного пакета Spark (хотя затем придется выполнить команду `spark-submit` или `pyspark` из каталога `bin` Spark).

Как работает SparkR

SparkR работает аналогично PySpark, но в настоящее время не позволяет пользователям выполнять произвольный код на языке R в работниках. Хотя аналогичный `PipedRDD` адаптер и существует для R, как и для Python, он остается закрытым, единственный общедоступный интерфейс для работы с R — через объекты `DataFrame`.

Среди непосредственно поддерживаемых языков SparkR наиболее далек от Scala фреймворка Spark в смысле набора возможностей. Этот разрыв, вероятно, со временем уменьшится, но если вы решили использовать SparkR, то обязательно убедитесь в наличии нужных функций. Эту информацию можно найти в документации по данному API (http://spark.apache.org/docs/latest/api/R/index.html).

Чтобы показать, как выглядит интерфейс SparkR, мы переписали наш стандартный пример с подсчетом слов на языке R (пример 7.8).

Пример 7.8. Подсчет слов с помощью SparkR

```
library(SparkR)

# Настройка SparkContext & SQLContext
sc <- sparkR.init(appName="high-performance-spark-wordcount-example")

# Инициализация SQLContext
sqlContext <- sparkRSQL.init(sc)

# Загрузка простых данных

df <- read.text(fileName)

# Разбиение по словам
words <- selectExpr(df, "split(value, \" \") as words")

# Подсчет количества
explodedWords <- select(words, alias(explode(words$words), "words"))
wc <- agg(groupBy(explodedWords, "words"), "words" = "count")

# Попытка завершается неудачей:
# resultingSchema <- structType(structField("words", "array<string>"))
# words <- dapply(df, function(line) {
#   y <- list()
#   y[[1]] <- strsplit(line[[1]], " ")
# }, resultingSchema)
# Не получается выполнить даже тождественное преобразование
# над DataFrame из read.text в версии Spark 2.0-preview
# (хотя с другими DataFrame работает без проблем).

# Отображаем результат
showDF(wc)
```

Выполнить собственный код на языке R позволит метод `dapply` объектов `DataFrame`, как показано в примере 7.9. Поддержка выполнения пользовательского кода в SparkR оставляет желать лучшего, как видно из проблем при попытке подсчитать количество слов с помощью `dapply` в примере 7.8.

Пример 7.9. Выполнение произвольного кода SparkR с помощью DataFrame

```
library(SparkR)

# Настройка SparkContext & SQLContext
sc <- sparkR.init(appName="high-performance-spark-wordcount-example")

# Инициализация SQLContext
sqlContext <- sparkRSQL.init(sc)

# Подсчет количества символов – обратите внимание,
# что это не работает на текстовых DataFrame из-за программной ошибки
df <- createDataFrame (sqlContext,
  list(list(1L, 1, "1"),
  list(2L, 2, "22"),
  list(3L, 3, "333")),
  c("a", "b", "c"))
resultingSchema <- structType(structField("length", "integer"))
result <- dapply(df, function(row) {
  y <- list()
  y <- cbind(y, nchar(row[[3]]))
}, resultingSchema)
showDF(result)
```

Внутренняя реализация метода `dapply` аналогична поддержке UDF в Python, но, поскольку API для наборов RDD не общедоступен, он предоставляет больше возможностей для дальнейших оптимизаций и поощряет разработку с помощью более оптимизированных API наборов `DataFrame`.

Как и в случае PySpark, произвольный не-JVM-код выполняется медленнее, чем обычный код Spark на языке Scala.

SparkR — не единственный интерфейс, обеспечивающий взаимодействие Spark и языка R. Широко используется и сторонняя библиотека Sparklyr (http://spark.rstudio.com/) от компании R Studio. С точки зрения производительности ее базовые механизмы при взаимодействии с JVM не отличаются от таковых пакета SparkR.

Spark.jl (Spark для языка программирования Julia)

Spark.jl (https://github.com/dfdx/Spark.jl) — один из самых свежих проектов по привязке Spark, а потому еще не поддерживает полнофункциональное подмножество API. Установка Spark.jl исключительно проста (пример 7.10), причем вместе с ним автоматически устанавливается поддерживаемая версия фреймворка Spark. Общая

архитектура Spark.jl напоминает архитектуру PySpark со своим, реализованным вне JVM классом `PipedRDD`, способным выполнять синтаксический разбор ограниченных объемов сериализованных данных из кода на языке Julia. Вышеприведенные предостережения по поводу производительности при использовании PySpark относятся и к Spark.jl.

Пример 7.10. Установка Spark для языка Julia

```
Pkg.clone("https://github.com/dfdx/Spark.jl")
Pkg.build("Spark")
# Понадобится также свежая версия ветки master JavaCall.jl
Pkg.checkout("JavaCall")
```

На момент написания данной книги полная сериализация именованных функций языка Julia невозможна, так что используемые в преобразованиях функции следует делать анонимными.

Пока в Spark.jl не поддерживаются операции над парами «ключ — значение», мы не можем даже написать простой образец подсчета слов. В частности, там нет операции `reduceByKey`, которая необходима для перетасовки. И хотя отсутствует и, например, функция `flatMap`, ее можно заменить операцией `mapPartitions`. Пока что Spark.jl еще только многообещающий проект, не готовый к реальному использованию.

Как работает Eclair JS

В проекте Eclair JS применяется другой подход: вместо поддержки R и Python Eclair JS большей частью не выходит за пределы JVM, за исключением драйверной программы. Eclair JS выполняет JavaScript как в JVM, так и в движке V8 JavaScript (https://developers.google.com/v8/), а функции в преобразованиях вычисляются JVM с помощью движка Nashorn (http://www.oracle.com/technetwork/articles/java/jf14-nashorn-2126515.html). Размежевание вычислений на стороне драйвера и работников дает возможность быстрой интеграции в рабочих узлах и использования привязок NodeJS (https://nodejs.org/en/) на драйвере. Схема приведена на рис. 7.4.

Этот в чем-то нетрадиционный подход означает, что в преобразованиях некоторые библиотечные функции могут оказаться недоступны, но избавляет нас от проблемы с двойной сериализацией, встречающейся в UDF PySpark и SparkR. Драйвер взаимодействует с Node с помощью Apache Toree (https://toree.incubator.apache.org/) для отправки нужных функций в JVM, которая затем отправляет их на рабочие узлы.

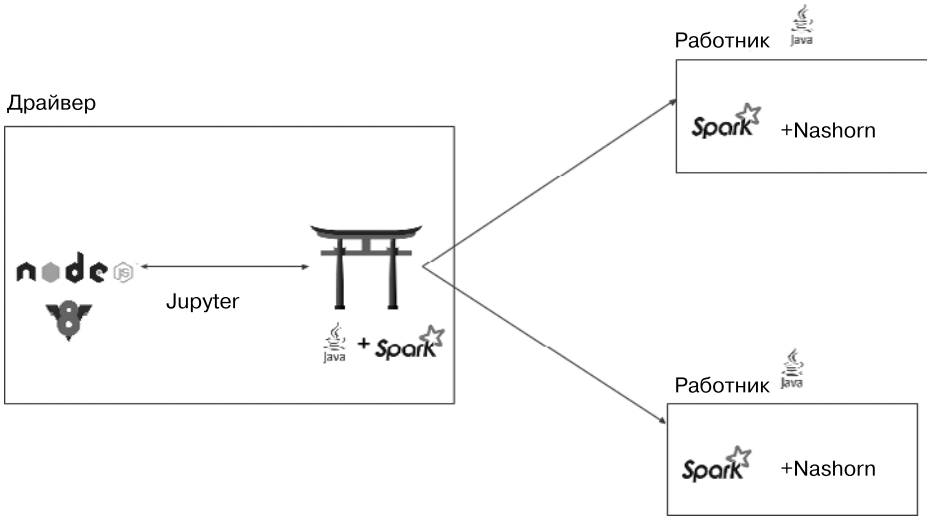

Рис. 7.4. Схема Eclair JS

Установка Eclair JS относительно проста по сравнению с другими языками программирования, так как для работы на стороне рабочих узлов не нужно никаких дополнительных пакетов. Процесс установки подробно описан в руководстве для начинающих (https://github.com/EclairJS/eclairjs-nashorn/wiki/Getting-Started-With-EclairJS-Nashorn).

Хотя некоторые из идей проекта Eclair JS были весьма интересны и оригинальны, он более не развивается и не рекомендуется к использованию.

Spark в среде CLR — C#

Проект Mobius (https://github.com/Microsoft/Mobius) компании Microsoft обеспечивает библиотеки привязками языка C# для работы с Apache Spark. Хотя общая архитектура аналогична архитектуре PySpark, внутренние компоненты `PythonRDD` взаимодействуют с общеязыковой исполняющей средой (common language runtime, CLR). Как и в случае с PySpark, преобразования наборов RDD включают копирование данных из JVM, а не использующие UDF на языке C# преобразования наборов `DataFrame` не требуют копирования данных на рабочих узлах (и вообще запуска CLR). Если вас заинтересовал Mobius, то загляните в его документацию (https://github.com/Microsoft/Mobius/tree/master/docs) и примеры (https://github.com/Microsoft/Mobius/tree/master/examples).

Обращение к коду на других языках программирования из Spark

Помимо использования других языков программирования для обращения к Spark, можно *вызывать* код на других языках из нашего фреймворка.

Использование pipe

Простейшее решение на случай, когда для языка, с которым вы работаете, не существует адаптеров, — применить интерфейс `pipe` фреймворка Spark. Для этого нужно сначала преобразовать наборы RDD в формат, подходящий для пересылки по Unix-конвейеру. Для взаимодействия часто используются простые форматы, например JSON или CSV, так как для генерации и разбора подобных записей на многих языках существуют облегченные библиотеки.

Вновь обратимся к примеру со Златовлаской из одноименного раздела на с. 151. Предположим, что, помимо оптимальной температуры овсяной каши панд, заказчик хочет выяснить, кто из панд комментировал рекламные сообщения Spark[1]; для этого можно «состряпать на скорую руку» небольшой сценарий на языке Perl наподобие того, что показан в примере. 7.11. Использовать данный сценарий в Spark поможет команда `pipe`, служащая для вызова этого сценария с рабочих узлов. Поскольку `pipe` работает только со строками, то придется преобразовать в них входные данные и выполнить синтаксический разбор полученной в результате строки в нужный тип данных, как показано в примере 7.12.

Пример 7.11. Сценарий на языке Perl, который мы будем вызывать из pipe

```
#!/usr/bin/perl
use strict;
use warnings;
use Pithub;
use Data::Dumper;

# Находим всех, кто оставлял комментарии в теме
my $user = $ENV{'user'};
my $repo = $ENV{'repo'};
my $p = Pithub->new(user => $user, repo => $repo);
while (my $id = <>) {
    chomp ($id);
    my $issue_comments = $p->issues->comments->list(issue_id => $id);
    print $id;
    while (my $comment = $issue_comments->next) { print " ".$comment->{"user"}-
          >{"login"};
    }
    print "\n";
}
```

[1] Это, прямо скажем, некоторое отступление от привычной сказки про Златовласку, но, надеемся, вы нас простите.

Пример 7.12. Используем интерфейс pipe (из кода Spark на языке Scala) для взаимодействия с программой на языке Perl в рабочих узлах

```
def lookupUserPRS(sc: SparkContext, input: RDD[Int]): RDD[(Int, List[String])] = {
  // Копируем сценарий на рабочие узлы с помощью вызова sc.addFile
  // Добавление файла требует указания абсолютных путей
  val distScriptName = "ghinfo.pl"
  val userDir = System.getProperty("user.dir")
  val localScript = s"${userDir}/src/main/perl/${distScriptName}"
  val addedFile = sc.addFile(localScript)

  // Передаем на рабочий узел переменные среды
  val enviromentVars = Map("user" -> "apache", "repo" -> "spark")
  val result = input.map(x => x.toString)
    .pipe(SparkFiles.get(distScriptName), enviromentVars)
  // Синтаксический разбор результатов
  result.map{record =>
    val elems: Array[String] = record.split(" ")
     (elems(0).toInt, elems.slice(1, elems.size).sorted.distinct.toList)
  }
}
```

Spark не скопирует автоматически сценарий на машины работников, так что при вызове пользовательской программы можно задействовать интерфейс sc.addFile, как в примере 7.12. В противном случае (скажем, при вызове системной программы) можете пропустить эту часть кода.

Как PySpark, так и SparkR используют специализированную версию наборов PipedRDD для обмена данными на рабочих узлах.

Проверьте, корректно ли обрабатываете вариант пустых секций, ведь вашу программу могут вызвать и для них (хотя эта функциональность может в следующих версиях поменяться).

JNI

Интерфейс прямого доступа к JVM (java native interface, JNI) — еще одна возможность стыковки с другими языками программирования. JNI позволяет успешно обращаться к некоторым библиотекам C/C++, как и к библиотекам других языков со статической компиляцией, например FORTRAN. Хотя JNI не испытывает таких же проблем с двойной сериализацией, как при обращении к PySpark или использовании интерфейса `pipe`, все равно придется копировать данные из JVM и обратно.

Именно поэтому некоторые библиотеки, такие как JBLAS (http://jblas.org/), реализуют часть компонентов в JVM, поскольку затраты на копирование сводят на нет все выгоды нативного кода.

Для иллюстрации использования JNI со Spark рассмотрим вызов очень простой функции на языке C, складывающей все ненулевые входные значения. Ее сигнатура приведена в примере 7.13.

Пример 7.13. Простой заголовочный файл на языке C

```
#ifndef _SUM_H
#define _SUM_H

int sum(int input[], int num_elem);

#endif /* _SUM_H */
```

Можно написать JNI-спецификации для вызова этой функции как из кода на языке Java (пример 7.14), так и из кода на Scala (пример 7.15). Хотя инструментальные средства Java несколько более просты, существенной разницы между ними нет.

Пример 7.14. Простая JNI-спецификация для Java

```
class SumJNIJava {
  public static native Integer sum(Integer[] array);
}
```

Пример 7.15. Простая JNI-спецификация для Scala

```
class SumJNI {
  @native def sum(n: Array[Int]): Int
}
```

Написание адаптеров вручную требует немалых усилий. Утилита SWIG может автоматически генерировать части необходимых привязок.

После написания функции на языке C и спецификации JNI-класса необходимо сгенерировать файлы `.class`, а на их основе — связующий заголовочный файл (пример 7.16). Команда `javah` генерирует на основе файлов `.class` заголовочные файлы, используемые затем для создания адаптера на стороне кода на языке C.

Пример 7.16. Генерация заголовочного файла с помощью интерфейса командной строки

```
javah -classpath ./target/examples-0.0.1.jar \
com.highperformancespark.examples.ffi.SumJNI
```

При формировании сборки с помощью SBT пакет `sbt-jni` Джейкоба Одерски (Jakob Odersky) упрощает интеграцию нативного кода с проектом на Scala. Пакет

`sbt-jni` опубликован в виде SBT-плагина аналогично `spark-packages-sbt`, включить его можно путем добавления записи в файл `project/plugins.sbt`, как показано в примере 7.17.

Пример 7.17. Добавление плагина sbt-jni в файл project/plugins.sbt

```
addSbtPlugin("ch.jodersky"  % % "sbt-jni"  % "1.0.0-RC3")
```

Пакет `sbt-jni` упрощает генерацию файла заголовка, добавляя цель `javah` в sbt, что приводит к генерации файлов заголовков и помещению их в каталог `./target/native/include/`.

После генерации файла заголовка необходимо написать адаптер на языке C. Менять сгенерированный файл заголовка не следует, лучше его импортировать в нашу оболочку, как показано в примере 7.18.

Пример 7.18. JNI-оболочка на C

```
#include "sum.h"
#include "include/com_highperformancespark_examples_ffi_SumJNI.h"
#include <ctype.h>
#include <jni.h>

/*
 * Класс:     com_highperformancespark_examples_ffi_SumJNI
 * Метод:     sum
 * Сигнатура: ([I)I
 */
JNIEXPORT jint JNICALL Java_com_highperformancespark_examples_ffi_SumJNI_sum
(JNIEnv *env, jobject obj, jintArray ja) {
  jsize size = (*env)->GetArrayLength(env, ja);
  jint *a = (*env)->GetIntArrayElements(env, ja, 0);
  return sum(a, size);
}
```

Пакет `sbt-jni` также упрощает создание и сборку нативного кода, добавляя пакеты `nativeCompile`, `javah` и `packageBin` с целью упростить компоновку JAR-файла с обоими нативными файлами и артефактами Java. Чтобы `sbt-jni` мог выполнять сборку и нативного кода (помимо JVM-кода), необходим сборочный файл. Если вы создаете новый проект, то для генерации файла-заготовки `CMakeLists.txt` (который можно будет использовать как основу для нативной сборки) необходимо указать sbt цель сборки `nativeInit CMake`.

В нашем примере проекта сборка нативного кода производится наряду с кодом на языке Scala. В качестве альтернативы можно создать отдельный пакет для нативного кода, особенно если планируется поддержка нескольких архитектур.

При сборке артефакта с помощью `sbt-jni` можно использовать декоратор `nativeLoader` из пакета `ch.jodersky.jni.nativeLoader` для автоматической загрузки нативного

кода по мере необходимости. В примере, над которым мы работаем, библиотека называется `libhigh-performance-spark0`, так что ее автоматическая загрузка обеспечивается путем добавления показанного в примере 7.19 декоратора в класс SumJNI.

Пример 7.19. Декоратор nativeLoader

```
@nativeLoader("high-performance-spark0")
```

При работе в языке Java или потребности в бо́льших возможностях управления пригодится метод `loadLibrary` класса `System`, который принимает в качестве параметра имя библиотеки и ищет ее в `java.library.path` или метод того же класса `load` с указанием абсолютного пути.

Убирайте префикс lib, автоматически добавляемый методом loadLibrary (и sbt-jni), чтобы не получать странные ошибки компоновки во время выполнения.

Удобно использовать в качестве справочника спецификации Oracle JNI (https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html).

Если нативная библиотека не включена в ваш JAR-файл, то необходимо убедиться, что JVM работника Spark может к ней обращаться. Если библиотека уже установлена на рабочих узлах, то можно добавить -Djava.library.path=... в spark.executor.extraJavaOptions.

Java Native Access (JNA)

Java Native Access (JNA) (https://github.com/java-native-access/jna) — альтернатива JNI от сообщества разработчиков, позволяющая вызывать нативный код, в идеале вообще без необходимого для JNI шаблонного кода. Хотя JNA — пакет, созданный сообществом, это не означает его низкого качества; он используется во множестве серьезных проектов и применялся разработчиками фреймворка Spark. Мы можем задействовать JNA для вызова предыдущего примера как в коде на Scala (пример 7.20), так и Java.

Пример 7.20. Простая JNA-спецификация для Scala

```
import com.sun.jna._
object SumJNA {
  Native.register("high-performance-spark0")
  @native def sum(n: Array[Int], size: Int): Int
}
```

Важно отметить, что приведенные образцы использования библиотеки JNA не требуют написания JNI-адаптера (как в примере 7.18), а непосредственно вызывают

функцию на языке C. Хотя с помощью SWIG можно легко сгенерировать много JNI-адаптеров, некоторые разработчики считают, что это неопровержимый довод в пользу применения JNA, а не JNI.

При использовании JNA существует возможность благодаря свойству jna.boot.library.path добавлять библиотеки в путь поиска до системного пути для библиотек.

Все еще FORTRAN

Поразительное количество библиотек для численных вычислений все еще содержит реализации на языке FORTRAN. К счастью, у многих из них уже есть Java- или Python-адаптеры, что намного упрощает обращение к ним. Зачастую эти библиотеки способны сами принимать разумные решения относительно того, для каких операций имеет смысл тратить ресурсы на копирование данных в FORTRAN, а какие операции лучше реализовать на другом языке. Не для всего кода на языке FORTRAN уже созданы адаптеры, и вы можете встретиться с необходимостью стыковки с подобным кодом.

Общая схема такова: сначала создать адаптер на C/C++, позволяющий вызывать код на FORTRAN из кода на Java, после чего скомпоновать код C/C++ вместе с кодом на FORTRAN. Если бы наш образец с суммированием был написан на языке FORTRAN (пример 7.21), то пришлось бы создать адаптер на C, такой как в примере 7.22, после чего пройти по шагам, описанным в подразделе «JNI» на с. 221, для вызова библиотеки на C.

Пример 7.21. Функция суммирования на FORTRAN

```
INTEGER FUNCTION SUMF(N,A) BIND(C, NAME='sumf')
INTEGER A(N)
SUMF=SUM(A)
END
```

Пример 7.22. Адаптер на C для написанной на FORTRAN функции суммирования

```
// Подпрограмма на FORTRAN
extern int sumf(int *, int[]);

// Вызов кода на языке FORTRAN, который должен находиться по ссылке size
int wrap_sum(int input[], int size) {
  return sumf(&size, input);
}
```

Если вам нравится пакет sbt-jni, то можете дополнить сгенерированный файл CMake для компиляции кода и на FORTRAN.

Такие программы, как fortrwrap (https://github.com/mcfarljm/fortwrap), позволяют автоматически генерировать подобные адаптеры. Или же их можно вообще опустить благодаря использованию JNA. Вызов написанной на языке FORTRAN функции с помощью JNA практически не отличается от вызова функции, написанной на языке C, как показано в примере 7.23.

Пример 7.23. Вызов написанной на FORTRAN функции SUMF с помощью JNA

```
import com.sun.jna._
import com.sun.jna.ptr._
object SumFJNA {
  Native.register("high-performance-spark0")
  @native def sumf(n: IntByReference, a: Array[Int]): Int
  def easySum(size: Int, a: Array[Int]): Int = {
    val ns = new IntByReference(size)
    sumf(ns, a)
  }
}
```

Вызывать код на FORTRAN из JVM сложнее, чем код на C. По возможности лучше использовать существующие адаптеры, способные оптимально выбрать, какие компоненты лучше выполнять на FORTRAN, а не в JVM.

Займемся GPU

GPU — еще один замечательный способ работы с параллельными задачами численных расчетов. Как выяснилось, они особенно эффективны в случае определенных типов задач машинного обучения. Смысл существования некоторых одноузловых распределенных систем состоит исключительно в координации работы нескольких GPU. Если ваша задача хорошо подходит для ускорения вычислений с помощью GPU, то можно добиться колоссального роста производительности (SparkGPULR (https://github.com/kiszk/spark-gpu/blob/dev/examples/src/main/scala/org/apache/spark/examples/SparkGPULR.scala) показывает трехкратный рост).

Пакет фреймворка Spark GPUEnabler (https://github.com/IBMSparkGPU/GPUEnabler) существует специально для упрощения сопряжения Spark с CUDA. Этот пакет упрощает установку JCUDA и автоматически преобразует данные в столбцовый формат для обработки на GPU.

Некоторые разработчики (https://iamtrask.github.io/2014/11/22/spark-gpu/) также используют фреймворк aparapi (https://github.com/aparapi/aparapi) для автоматизации компиляции Java-кода в OpenCL (https://en.wikipedia.org/wiki/OpenCL), хотя пакетов, упрощающих эту интеграцию, в настоящий момент не существует.

На сегодня в Apache Spark нет единого способа ускорения вычислений с помощью GPU. Среди предложенных вариантов — пакет spark-gpu (https://github.com/kiszk/spark-gpu/) от компании IBM, пакет spark-gpu (https://github.com/adobe-research/spark-gpu) от Adobe и др.

Если вас заинтересовал этот вопрос, то загляните в предложение SPARK-12620 (https://issues.apache.org/jira/browse/SPARK-12620) и ему подобные.

Что нас ждет в будущем

Компонент Tungsten способен обеспечить для фреймворка Spark хранение данных вне кучи, но формат данных в настоящее время нестабилен и недостаточно документирован для совместного доступа из кода на других языках. Существуют две возможности решения этой проблемы: стандартизация Tungsten, SPARK-9697 (https://issues.apache.org/jira/browse/SPARK-9697) или интеграция Arrow в Python и Spark, SPARK-13534 (https://issues.apache.org/jira/browse/SPARK-13534). Надеемся, в будущих изданиях книги мы сможем с радостью констатировать, что эти изменения были воплощены в жизнь.

Резюме

Написание высокопроизводительного кода для фреймворка Spark не должно ограничиваться языком Scala, не говоря уже о JVM (хотя это многое упростило бы). В Spark существует множество библиотек привязок для различных языков программирования, как встроенных, так и сторонних, а JNI, JNA, конвейеры и сокеты помогут взаимодействовать с еще большим количеством языков. Для некоторых операций затраты на копирование данных из JVM и обратно могут превышать затраты на ее выполнение в JVM — даже при использовании специализированных библиотек, — так что важно взвесить сложность ваших преобразований, прежде чем выходить за пределы JVM. В данный момент не поддерживаемая стандартизация поддержки хранения данных вне кучи компонентом Tungsten со временем может улучшить языковую совместимость на рабочих узлах.

8 Тестирование и валидация

Автоматическое тестирование в мире Spark часто недооценивают, однако ручная проверка функциональности занимает много времени и чревата ошибками, особенно в случае работающих длительное время пакетных заданий и сложных потоковых архитектур. Эффективные тесты ускоряют разработку и упрощают рефакторинг, проводимый для повышения производительности.

Тесты для проверки производительности требуют дополнительных усилий, особенно в распределенных системах. Однако с помощью счетчиков Spark можно получить статистику по времени выполнения от всех работников, количество обработанных и перетасованных записей. Эти счетчики служат той же цели, что и системные хронометражи в системе из одной машины.

Тестирование — отличный способ обнаружить ошибки, которые можно вообразить. Но на практике часто возникают новые и захватывающие виды сбоев программного обеспечения, иногда далеко не столь очевидные, как исключение нулевого указателя. В подобных случаях важна сама возможность выявить факт ошибки, чтобы избежать принятия решений на основе ошибочных моделей.

Модульное тестирование

Модульное тестирование позволяет сосредоточиться на тестировании маленьких компонентов свойств со сложными зависимостями (например, источников данных) зачастую с помощью замены заглушками. Такие тесты обычно выполняются быстрее, чем комплексные, а потому часто используются при разработке. При необходимости рефакторинга можно протестировать большую часть кода без учета каких-либо особенностей Spark. Процесс тестирования же остальной части кода существенно упрощают библиотеки.

Основное модульное тестирование в Spark

Задание Spark может быть написано так, что самые мелкие его компоненты будут тестироваться и без каких-либо зависимостей фреймворка. Для тестирования потока данных самого задания Spark понадобится объект `SparkContext`, чтобы создать тестовые наборы RDD или потоки DStream с локальными коллекциями. Из этого объекта можно применить требуемые преобразования, сравнивая локально полученные результаты с помощью выбранного фреймворка модульного тестирования.

Разбиение кода на элементарные операции в целях тестирования

Поскольку многие преобразования Spark принимают в качестве аргументов функции, работающие с отдельными элементами или итераторами секций, то можно протестировать эти функции (пример 8.1), не прибегая к созданию набора RDD или использованию объекта `SparkContext` (в отличие от примера 8.2).

Пример 8.1. Простая в тестировании функция-аргумент

```
def tokenizeRDD(input: RDD[String]) = {
  input.flatMap(tokenize)
}

protected[tokenize] def tokenize(input: String) = {
  input.split(" ")
}
```

Пример 8.2. Сложная в тестировании функция-аргумент

```
def difficultTokenizeRDD(input: RDD[String]) = {
  input.flatMap(_.split(" "))
}
```

Возвращаясь к нашему примеру со Златовлаской, мы видим, что подобным образом можно протестировать компонент, извлекающий из секции значения для заданных индексов (см. пример 6.25). При таком разбиении кода будет уместно обычное хорошо знакомое модульное тестирование. Удобочитаемость — еще один веский довод избегать синтаксиса анонимных функций языка Scala в случае слишком сложных функций.

Даже если у вас есть возможность разбить свои вспомогательные функции для раздельного тестирования внутренней логики преобразований, все равно не помешает выполнить тестирование на наборах RDD или потоках данных DStream, чтобы обнаружить потенциальные ошибки сериализации.

Стандартные задания Spark (тестирование с RDD)

Помимо тестирования передаваемых в преобразования функций, важно проверить и выражаемую этими преобразованиями логику. Простейший способ тестирования преобразований — создать объект `SparkContext`, распараллелить входные данные, применить преобразования и выполнить локальный сбор результатов для сравнения с ожидаемыми значениями (пример 8.3).

Пример 8.3. Простой модульный тест для Spark

```
class QuantileOnlyArtisanalTest extends FunSuite with BeforeAndAfterAll {
  @transient private var _sc: SparkContext = _
  def sc: SparkContext = _sc

  val conf = new SparkConf().setMaster("local[4]").setAppName("test")

  override def beforeAll() {
    _sc = new SparkContext(conf)
    super.beforeAll()
  }

  val inputList = List(GoldiLocksRow(0.0, 4.5, 7.7, 5.0),
    GoldiLocksRow(4.0, 5.5, 0.5, 8.0),
    GoldiLocksRow(1.0, 5.5, 6.7, 6.0),
    GoldiLocksRow(3.0, 5.5, 0.5, 7.0),
    GoldiLocksRow(2.0, 5.5, 1.5, 7.0)
  )

  val expectedResult = Map[Int, Set[Double]](
    0 -> Set(1.0, 2.0),
    1 -> Set(5.5, 5.5),
    2 -> Set(0.5, 1.5),
    3 -> Set(6.0, 7.0))

  test("Наивное решение задачи Златовласки"){
    val sqlContext = new SQLContext(sc)
    val input = sqlContext.createDataFrame(inputList)
    val whileLoopSolution = GoldilocksWhileLoop.findRankStatistics(
      input, List(2L, 3L)).mapValues(_.toSet)
    val inputAsKeyValuePairs = GoldilocksGroupByKey.mapToKeyValuePairs(input)
    val groupByKeySolution = GoldilocksGroupByKey.findRankStatistics(
      inputAsKeyValuePairs, List(2L,3L)).mapValues(_.toSet)
    assert(whileLoopSolution == expectedResult)
    assert(groupByKeySolution == expectedResult)
  }

  override def afterAll() {
    // Очищаем значение порта драйвера, чтобы случайно не попытаться
    // подключиться к тому же порту при перезагрузке
    sc.stop()
    System.clearProperty("spark.driver.port")
    _sc = null
    super.afterAll()
  }
}
```

Неожиданный нюанс в примере 8.3 — очистка значения для порта: это необходимо, чтобы при запуске нескольких тестов один за другим они не подключались к одному порту, так как попытка подключиться к уже используемому порту приведет к генерации исключения.

Если объем данных настолько велик, что непосредственно вызвать метод `collect` нельзя, то можно применить метод `toLocalIterator`, возвращающий по одной секции за раз. В случае процесса, создающего очень большой набор RDD, можно использовать методики сравнения наборов (описываемые в пункте «Вычисление различий наборов RDD» на с. 238 и в разделе «Комплексное тестирование» на с. 241). Однако последнее действие может быть признаком того, что объем тестовых данных превысил размеры, допустимые для модульного тестирования.

Метод toLocalIterator может инициировать несколько вычислений, чтобы гарантировать кэширование или сохранение набора RDD, с которым он работает.

Потоковая обработка

Чтобы протестировать приложения, использующие Spark Streaming, необходимо отдельно создать тестовые потоки, произвести локальный сбор данных для проверки соответствия результатов ожидаемым и выяснить, завершено ли выполнение теста. Если это подходит для вашего приложения, то библиотека spark-testing-base предоставляет адаптер, благодаря которому для создания теста достаточно задать входные данные и ожидаемый результат.

Быстро создать входные потоки данных поможет API `queueStream` (пример 8.4). Однако полученные в результате потоки (начиная со Spark 1.4.1) не поддерживают операции, для которых необходимо создание контрольных точек (такие как оконные функции или метод `updateStateByKey`). Более функциональный тестовый поток можно обеспечить, создав пользовательский объект `InputDStream` или применив классы `fileStream` либо `rawSocketStream` и записав входные данные в локальную файловую систему или сокет. Класс `TestInputStream` из библиотеки spark-testing-base ведет себя как версия класса `queueStream`, поддерживающая создание контрольных точек (пример 8.5).

Пример 8.4. Создание входного потока DStream, не поддерживающего создание контрольных точек

```
def makeSimpleQueueStream(ssc: StreamingContext) = {
  val input = List(List("привет"), List("счастливые панды", "грустные панды"))
    .map(sc.parallelize(_))
  val idstream = ssc.queueStream(Queue(input:_*))
}
```

Пример 8.5. Создание входного потока DStream, поддерживающего создание контрольных точек, с помощью библиотеки spark-testing-base

```
/**
 * Создаем входной поток на основе полученной входной последовательности.
 * При этом используется тип TestInputStream, поскольку тип queueStreams
   не поддерживает создание контрольных точек.
 */
private[holdenkarau] def createTestInputStream[T: ClassTag](
    sc: SparkContext,
    ssc_ : TestStreamingContext,
    input: Seq[Seq[T]]): TestInputStream[T] = {
  new TestInputStream(sc, ssc_, input, numInputPartitions)
}
```

По сравнению с созданием входных данных собрать результаты потоковой обработки нетрудно. Простейшее решение — использовать нечто подобное объекту `ArrayBuffer` с операцией `foreachRDD` (пример 8.6).

Пример 8.6. Сбор результатов

```
class TestOutputStream[T: ClassTag](parent: DStream[T],
  val output: ArrayBuffer[Seq[T]] = ArrayBuffer[Seq[T]]()) extends Serializable {

  parent.foreachRDD{(rdd: RDD[T], time) =>
    val collected = rdd.collect()
    output += collected
  }

}
```

Больше сложностей доставляет определение момента завершения потокового теста. Простейший подход — дождаться момента совпадения количества собранных результатов с ожидаемым значением, используя в качестве запасного варианта превышение времени ожидания. Однако при неудачном выборе времени ожидания это способно привести к ненадежному тесту (пример 8.7). Для управления работой Spark Streaming можно создать часы вручную, но данное решение требует расширения многих внутренних классов, которые могут поменяться в следующих версиях.

Пример 8.7. «Кустарный» тест потоковой обработки (ненадежный и не поддерживающий оконные и тому подобные операции)

```
test("кустарный потоковый тест") {
  val ssc = new StreamingContext(sc, Seconds(1))
  val input = List(List("привет"), List("счастливые панды", "грустные панды"))
      .map(sc.parallelize(_))
  // Примечание: не подходит для оконных операций
  // или создания контрольных точек
  val idstream = ssc.queueStream(Queue(input:_*))
  val tdstream = idstream.filter(_.contains("панды"))
  val result = ArrayBuffer[String]()
  tdstream.foreachRDD{(rdd: RDD[String], _) =>
    result ++= rdd.collect()
  }
  val startTime = System.currentTimeMillis()
  val maxWaitTime = 60 * 60 * 30
```

```
  ssc.start()
  while (result.size < 2 && System.currentTimeMillis() - startTime < maxWaitTime) {
    ssc.awaitTerminationOrTimeout(50)
  }
  ssc.stop(stopSparkContext = false)
  assert(List("счастливые панды", "грустные панды") === result.toList)
}
```

Чтобы упростить работу, библиотека spark-testing-base предоставляет два базовых класса для тестирования потоковой обработки: `StreamingSuiteBase` для преобразований и `StreamingActionBase` — для действий. Это позволяет писать тесты для преобразований путем указания ожидаемых входных данных и результатов (пример 8.8), в то время как библиотека сама выполняет стыковку с внутренними компонентами.

Пример 8.8. Вариант теста с использованием класса StreamingSuiteBase

```
test("очень простое преобразование") {
  val input = List(List("привет"), List("привет холден"), List("пока"))
  val expected = List(List("привет"), List("привет", "холден"), List("пока"))
  testOperation[String, String](input, tokenize _, expected, ordered = false)
}

// Пример тестируемой функции
def tokenize(f: DStream[String]): DStream[String] = {
  f.flatMap(_.split(" "))
}
```

Имитационное моделирование наборов RDD

Тестирование с помощью объекта `SparkContext` ведет к дополнительным расходам ресурсов, поскольку требует создания локального кластера Spark. Эти расходы могут замедлить выполнение тестов, особенно при использовании таких фреймворков, как scalacheck, генерирующих сотни тестов. В случае тестов, предназначенных для тестирования бизнес-логики, а не нюансов взаимодействия со Spark (например, сериализуемости), имитационные наборы RDD позволяют выполнять больше тестов и намного быстрее.

Хотя тестирование с помощью имитационных наборов RDD — отличный способ проверки бизнес-логики, тестирование будет неполным, поскольку при нем не проверяются сериализация и другие взаимодействия Spark. Даниэль Вестхайде (Daniel Westheide) создал библиотеку kontextfrei (https://github.com/dwestheide/kontextfrei), поддерживающую немало операций над имитационными наборами RDD.

Библиотека kontextfrei предназначена только для языка Scala и позволяет писать код бизнес-логики и тестов для приложений Spark без использования наборов RDD, но с тем же API. Ради быстрого получения обратной связи при разработке можно выполнить эти тесты и над быстрыми коллекциями языка Scala, и над наборами на сервере непрерывной интеграции, чтобы убедиться в отсутствии проблем с сериализацией.

Тестирование объектов DataFrame. Основная сложность при тестировании этих объектов — выяснить, совпадает ли полученный результат с ожидаемым значением. В простейшем случае, если нас интересует только несколько столбцов, можно просто извлечь элементы и сравнить их. Но при сложных объектах `DataFrame` такое сравнение быстро становится утомительным занятием, поэтому уместно сравнивать также с ожидаемыми строками (пример 8.9).

Пример 8.9. Сравнение объектов DataFrame по равенству строк

```
test("проверяем на точное равенство") {
  // Тест minHappyPandas
  val inputDF = sqlContext.createDataFrame(pandaInfoList)
  val result = HappyPandas.minHappyPandas(inputDF, 2)
  val resultRows = result.collect()

  val expectedRows = List(Row(sandiego, "красная", 2, 3))
  assert(expectedRows === resultRows)
}
```

Сравнение строк чаще всего срабатывает, но в случае объектов ByteArrays простого сравнения на равенство объектов Scala недостаточно. В этом случае необходимо сравнивать значения.

Поскольку в объектах `DataFrame` часто содержатся данные с плавающей точкой, то одной проверки на равенство может оказаться недостаточно. Здесь пригодится метод `approxEqualDataFrames` из библиотеки spark-testing-base, предназначенный для приближенного сравнения двух объектов `DataFrame` (пример 8.10) или сравнения отдельных элементов, как в примере 8.11.

Пример 8.10. Приближенное сравнение двух объектов DataFrame

```
test("проверяем процентное содержание счастливых панд") {
  val expectedList = List(Row(toronto, 0.5),
    Row(sandiego, 2/3.0),
    Row(virginia, 1/10.0))
  val expectedDf = createDF(expectedList, ("место", StringType),
                                          ("процентСчастливых", DoubleType))

  val inputDF = sqlContext.createDataFrame(pandaInfoList)
  val resultDF = HappyPandas.happyPandasPercentage(inputDF)

  assertDataFrameApproximateEquals(expectedDf, resultDF, 1E-5)
}
```

Пример 8.11. Приближенное сравнение с помощью библиотеки Scalatest и применение типажа +– к элементам строк

```
  assert(expectedRows.length === resultRows.length)
  expectedRows.zip(resultRows).foreach{case (r1, r2) =>
    assert(r1(0) === r2(0))
    assert(r1.getDouble(1) === (r2.getDouble(1) +- 0.001))
  }
```

Помимо проверки на равенство, необходимо также создать объект `SQLContext` для нашего набора тестов. Сформировать объект `SQLContext` можно в методе `beforeAll` аналогично тому, как мы поступали со SparkContext в примере 8.3.

Получение тестовых данных

До сих пор мы фокусировались на простых небольших наборах тестовых данных, создаваемых вручную. Это неплохо подходит для базового модульного тестирования, но в Spark многие ошибки проявляются лишь на больших наборах данных. Без достаточного объема (разнообразных) данных не обнаружить несбалансированное секционирование, неправильную обработку пустых секций и другие проблемы.

Идеальный источник тестовых данных — выборка из реальных данных, хотя это не всегда возможно вследствие правовых аспектов и соображений конфиденциальности. Выборка из реальных данных — во многих смыслах «золотой стандарт»: не приходится волноваться, что сгенерированные данные окажутся нерепрезентативными.

Если тестирование нужно выполнить на наборе данных, который слишком велик для генерации или хранения на отдельной машине, то можно применить настраиваемые генераторы случайных распределенных наборов RDD библиотеки MLlib. При использовании этих генераторов полезно по крайней мере попытаться получить соответствующее реальным данным распределение, чтобы при проверке обнаружить любые возможные проблемы, связанные с асимметрией данных.

Помимо генерации больших наборов данных явным образом, существует и другая методика: тестирование на основе свойств (property-based testing), при котором задаются инварианты кода, а генерация тестовых входных данных делегируется подпрограмме проверки свойств (property checker). При разумном проектировании подобные подпрограммы могут с помощью встроенных инструментов фреймворка Spark генерировать по-настоящему масштабные наборы данных для тестирования.

Генерация больших наборов данных

Зачастую при поиске причин проблем с производительностью для тестирования требуются большие наборы данных. А поскольку причина проблем часто заключается в асимметрии ключей или данных, важно понимать, какое распределение должно быть у генерируемых данных, как обсуждалось в главе 6.

Вернемся к примеру со Златовлаской. Если наша овсяная каша или любые другие рыночные данные распределяются по почтовому индексу, то в конце концов окажется, что на одни индексы приходится гораздо больше данных, чем на другие.

Во фреймворке Spark в объекте `RandomRDDs` (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.random.RandomRDDs$) библиотеки `mllib` имеются встроенные компоненты для генерации случайных наборов RDD. В нем есть встроенные функции-генераторы для экспоненциального, гамма-, логарифмического,

нормального, пуассоновского и равномерного распределений как для наборов RDD чисел типа `double`, так и для наборов RDD векторов. При наличии данных с несколькими распределениями (как в задаче Златовласки) можно сгенерировать различные компоненты и соединить их, что и представлено в примере 8.12. В случае более сложных типов данных или других распределений можно реализовать собственный генератор элементов, расширив класс `RandomDataGenerator`.

В более новых версиях библиотеки spark-testing-base на основе объекта RandomRDDs генерируются наборы данных, слишком большие для локальной машины. Некоторые другие библиотеки проверки свойств пока еще не поддерживают эту возможность.

Пример 8.12. Генерация данных с целью масштабирования производительности для примера со Златовлаской

```
/**
 * Генерируем данные Златовласки с одним и тем же ID.
 * Ожидается, что почтовые индексы подчиняются экспоненциальному
 * распределению, а сами данные – нормальному.
 * Упрощено, чтобы избежать тройной операции zip.
 *
 * Примечание: количество сгенерированных строк может оказаться меньше
 * запрошенного вследствие различных распределений в разных секциях
 * при вычислении почтового индекса по секциям.
 */
def generateGoldilocks(sc: SparkContext, rows: Long, numCols: Int):
    RDD[RawPanda] = {
  val zipRDD = RandomRDDs.exponentialRDD(sc, mean = 1000,  size = rows)
    .map(_.toInt.toString)
  val valuesRDD = RandomRDDs.normalVectorRDD(
    sc, numRows = rows, numCols = numCols)
  zipRDD.zip(valuesRDD).map{case (z, v) =>
    RawPanda(1, z, "большая", v(0) > 0.5, v.toArray)
  }
}
```

Проект spark-sql-perf (https://github.com/databricks/spark-sql-perf) от компании Databricks — еще один образец генерации больших наборов данных для тестирования. С помощью ScalaCheck можно также создать фиктивный тест, чтобы сохранять данные для использования их в будущем.

Выборка

Если это возможно в вашем случае, то выборка реальных данных — отличный источник тестовых данных. Функциональность как наборов RDD Spark Core, так и наборов RDD пар «ключ — значение» поддерживает настраиваемые случайные выборки. Когда работа зависит от соединений нескольких таблиц, важно, чтобы таблица с выбранными данными содержала все нужные записи.

Простейший способ выборки, включенный прямо в класс RDD, — функция `sample` со следующими параметрами: `withReplacement: Boolean, fraction: Double, seed: Long (optional)` (пример 8.13). Благодаря им можно указать, какая часть данных нужна, и осуществить выборку без особых проблем.

Пример 8.13. Выборка из обычного RDD

```
rdd.sample(withReplacement=false, fraction=0.1)
```

Размеры итоговой выборки задаются в виде доли от входных данных: в случае надобности в повышающей дискретизации этот параметр можно указать >1, если withReplacement=true.

В зависимости от того, выполняется ли выборка с возвращением, метод `sample` формирует объект `PartitionwiseSampledRDD` с двумя различными методами получения выборки (Пуассона или Бернулли). Если требуется больше возможностей настройки, то можно непосредственно создать объект `PartitionwiseSampledRDD` с нужным методом выборки при условии реализации им типажа `RandomSampler` из пакета `org.apache.spark.util.random`.

Выполнение выборки может понадобиться и при машинном обучении. Иногда способна пригодиться выборка из множества X (то есть часть элементов множества X) и ее дополнение (множество всех не входящих в нее элементов), и в этом случае благодаря созданию вручную объекта PartitionwiseSampledRDD появляется возможность использовать функцию cloneComplement() для формирования дополнения набора PartitionwiseSampledRDD.

При работе с данными из нескольких классов часто нужна гарантия, что в выборке представлены данные из каждого класса. Методы `sampleByKeyExact` и `sampleByKey` принимают на входе ассоциативный массив с процентным соотношением для каждого из ключей, благодаря чему становится возможно выполнение стратифицированной выборки (пример 8.14). В зависимости от требуемых аспектов можно сделать так, чтобы выборка отражала распределение ключей во входных данных или генеральной совокупности, попытаться создать сбалансированную выборку из несбалансированной генеральной совокупности (HAM/SPAM) или получить определенный процент пользователей из разных мест с целью проверить, корректно ли они обрабатываются.

Пример 8.14. Стратифицированная выборка

```
// 5 % красных панд и 50 % гигантских панд
val stratas = Map("красная" -> 0.05, "гигантская" -> 0.50)
rdd.sampleByKey(withReplacement=false, fractions = stratas)
```

Иногда возврата выборки в одном наборе RDD недостаточно и нужно одновременно выполнять выборку в несколько различных наборов. Функция `randomSplit` принимает на входе массив весов, а возвращает массив наборов RDD с элементами пропорционально этим весам.

В наборах `DataFrame` доступны функции `sample` и `randomSplit`. При необходимости выполнить стратифицированную выборку из набора `DataFrame` сначала нужно преобразовать его в набор RDD.

Проверка свойств с помощью библиотеки ScalaCheck

ScalaCheck (https://www.scalacheck.org/) — библиотека тестирования на основе свойств для языка Scala, подобная библиотеке QuickCheck языка Haskell (https://hackage.haskell.org/package/QuickCheck). Тестирование на основе свойств позволяет задать для кода инварианты (допустим, во всех результатах должна содержаться подстрока «панда»), а библиотека тестирования сгенерирует различные виды тестовых данных. Две библиотеки, sscheck (https://github.com/juanrh/sscheck) и spark-testing-base (https://github.com/holdenk/spark-testing-base), реализуют генераторы для фреймворка Spark. Проверка свойств с помощью библиотеки spark-testing-base продемонстрирована в примере 8.15.

Пример 8.15. Тестирование на основе свойств с помощью ScalaCheck

```
// Простое свойство: отображение не должно менять количество элементов
test("отображение не должно менять количество элементов ") {
  val property =
    forAll(RDDGenerator.genRDD[String](sc)(Arbitrary.arbitrary[String])) {
      rdd => rdd.map(_.length).count() == rdd.count()
    }

  check(property)
}
```

Библиотека ScalaCheck автоматически генерирует множество часто встречающихся граничных условий, но можно также указать входные данные, которые, по нашему мнению, способны вызвать проблемы. Сам генератор создает наборы RDD с различными размерами секций с тенденцией к созданию определенного количества пустых секций.

Вычисление различий наборов RDD. Локальный сбор данных работает неплохо, если тестовые данные достаточно невелики для того, чтобы их можно было собрать на локальную машину. В случае выполнения тестов в кластере при большом наборе данных приходится вычислять разницу между наборами RDD — без передачи всех данных обратно в кластер.

Большую часть модульных тестов можно выполнить с помощью одних лишь методов `parallelize` и `collect`; однако при сравнении наборов RDD, как и сравнении

коллекций, есть несколько вариантов. Мы рассмотрим два случая: когда порядок важен и когда несущественен.

Начнем со сравнения двух наборов RDD в случае, когда предполагается одинаковая их упорядоченность (пример 8.16). Это возможно при вызове метода `sortByKey`, подобно тому как мы поступили в задаче Златовласки после подсчета уникальных значений. В случае одинаковых объектов `Partitioner` можно объединить два RDD с помощью операции `zip` и сравнить элементы непосредственно, без перетасовки. В противном случае можно просто секционировать один из RDD заново так, чтобы он соответствовал второму.

Пример 8.16. Сравнение наборов RDD

```
/**
 * Утверждается, что два набора RDD равны (при одинаковом упорядочении).
 * Если они равны — утверждение выполняется успешно,
 * если нет — завершается ошибкой.
 */
def assertRDDEqualsWithOrder[T: ClassTag](
  expected: RDD[T], result: RDD[T]): Unit = {
  assertTrue(compareRDDWithOrder(expected, result).isEmpty)
}

/**
 * Сравниваем два набора RDD с учетом упорядоченности
 * (например, [1,2,3] != [3,2,1]).
 * В случае различных объектов Partitioner для этого потребуется
 * несколько проходов по исходным данным.
 * В случае равенства возвращается None, в противном случае возвращается
 * Some с первым различающимся элементом.
 * Если длины не совпадают, то один из двух компонентов может быть равен None.
 */
def compareRDDWithOrder[T: ClassTag](
  expected: RDD[T], result: RDD[T]): Option[(Option[T], Option[T])] = {
  // При наличии известного объекта Partitioner просто выполняем
  // операцию zip.
  if (result.partitioner.map(_ == expected.partitioner.get).getOrElse(false)) {
    compareRDDWithOrderSamePartitioner(expected, result)
  } else {
    // В противном случае индексируем все элементы.
    def indexRDD[T](rdd: RDD[T]): RDD[(Long, T)] = {
      rdd.zipWithIndex.map { case (x, y) => (y, x) }
    }
    val indexedExpected = indexRDD(expected)
    val indexedResult = indexRDD(result)
    indexedExpected.cogroup(indexedResult).filter { case (_, (i1, i2)) =>
      i1.isEmpty || i2.isEmpty || i1.head != i2.head
    }.take(1).headOption.
      map { case (_, (i1, i2)) =>
         (i1.headOption, i2.headOption) }.take(1).headOption
  }
}

/**
 * Сравниваем два набора RDD. В случае равенства возвращается None,
```

```
 * в противном случае возвращается Some с первым различающимся элементом.
 * Считаем, что объекты Partitioner одинаковы.
 */
def compareRDDWithOrderSamePartitioner[T: ClassTag](
  expected: RDD[T], result: RDD[T]): Option[(Option[T], Option[T])] = {
  // В случае различных длин элементов преобразуем в объекты Option
  // и заполняем None.
  expected.zipPartitions(result) {
    (thisIter, otherIter) =>
      new Iterator[(Option[T], Option[T])] {
        def hasNext: Boolean = (thisIter.hasNext || otherIter.hasNext)

        def next(): (Option[T], Option[T]) = {
           (thisIter.hasNext, otherIter.hasNext) match {
            case (false, true) => (Option.empty[T], Some(otherIter.next()))
            case (true, false) => (Some(thisIter.next()), Option.empty[T])
            case (true, true) => (Some(thisIter.next()), Some(otherIter.next()))
            case _ => throw new Exception("Вызов next после того, как элементы
                                          закончились")
          }
        }
      }
  }.filter { case (v1, v2) => v1 != v2 }.take(1).headOption
}
```

Если порядок не имеет значения и достаточно оператора сравнения на равенство, то можно совместно сгруппировать два набора RDD (вместе с фиктивными значениями), как показано в примере 8.17.

Пример 8.17. Сравнение наборов RDD без учета порядка

```
/**
 * Утверждается, что два набора RDD равны (без учета упорядоченности).
 * Если они равны — утверждение выполняется успешно,
 * если нет — завершается ошибкой.
 */
def assertRDDEquals[T: ClassTag](expected: RDD[T], result: RDD[T]): Unit = {
  assertTrue(compareRDD(expected, result).isEmpty)
}

/**
 * Сравниваем два набора RDD, не требуя, чтобы они были упорядочены
 * одинаковым образом. В случае равенства возвращается None,
 * в противном случае возвращается Some с первым различающимся элементом
 *
 * @возвращает None, если два RDD равны, или Some, содержащий информацию
 *              о первом различающемся элементе.
 *              Информация о различии представляет собой кортеж
 *              из трех элементов: (ключ, число вхождений ключа в ожидаемый
 *              набор RDD, число вхождений ключа в итоговый набор RDD).
 */
def compareRDD[T: ClassTag](expected: RDD[T], result: RDD[T]):
    Option[(T, Int, Int)] = {
  // Отображение и подсчет количеств для всех уникальных элементов.
  val expectedKeyed = expected.map(x => (x, 1)).reduceByKey(_ + _)
```

```
  val resultKeyed = result.map(x => (x, 1)).reduceByKey(_ + _)
  // Группируем и фильтруем различия.
  expectedKeyed.cogroup(resultKeyed).filter { case (_, (i1, i2)) =>
    i1.isEmpty || i2.isEmpty || i1.head != i2.head
  }
    .take(1).headOption.
    map { case (v, (i1, i2)) =>
       (v, i1.headOption.getOrElse(0), i2.headOption.getOrElse(0)) }
}
```

Эти примеры, вероятно, не отражают в точности то, как необходимо проверять равенство, но при желании их можно обобщить. Сравнивая, скажем, наборы RDD чисел типа `double`, можно сначала их отсортировать, а затем воспользоваться функцией, аналогичной `compareWithOrder`, но проверять с неким допуском, а не устанавливать точное равенство.

Тот же общий подход работает и для вычисления разности наборов DataFrame, он реализован в библиотеке spark-testing-base в методах equalDataFrames и approxEqualDataFrames.

Комплексное тестирование

Иногда для тестирования системы недостаточно простого создания фиктивных источников данных и имитационного моделирования (или это оказывается слишком сложной задачей). Комплексное тестирование и выяснение причин отказов в момент его выполнения требует больше времени, чем модульное, однако при этом выявляются совершенно другие типы ошибок. Комплексное тестирование можно комбинировать с некоторыми из вышеупомянутых методов проверки производительности, что позволит оценить производительность не только фрагментов кода, но и системы в целом.

Выбор среды комплексного тестирования

Рано или поздно наступает предел тому, что можно проверить с помощью модульных тестов. В масштабных информационных системах распространенным источником ошибок являются ошибки при получении, внесении и обработке данных для последующего использования, а также разногласия в понимании форматов. Поэтому комплексное тестирование системы важно — хотя и представляет собой более сложную задачу, чем модульное.

Локальный режим тестирования. Хотя мы говорили о локальном режиме тестирования в основном в контексте модульного тестирования (см. раздел «Модульное тестирование» на с. 228), для достаточно небольших проектов при комплексном тестировании можно использовать немало аналогичных методик. Вместо создания

фиктивных источников данных и применения таких функций, как `parallelize` или `queueStream`, можно задействовать уменьшенные версии наших хранилищ или потоковых источников данных.

С помощью Docker. Контейнеры Docker — удобный способ компоновки и распространения облегченных виртуальных контейнеров, существенно упрощающий автоматическое развертывание. Они особенно удобны для комплексного тестирования в случае распределенных систем, подобных Spark, для которых требуются и работники, и ведущий узел, а также различные необходимые для ваших данных сервисы (например, HDFS, Mongo и т. д.). Существует несколько образцов проектов по настройке среды комплексного тестирования Spark на основе Docker, включая созданный @cfregley проект pipeline (https://github.com/PipelineAI/pipeline/wiki) и проект spark-perf компании Databricks (https://github.com/databricks/spark-perf).

Мини-кластеры Yarn. Встроенные в Hadoop библиотеки тестирования позволяют настраивать локальный кластер Yarn еще более простым способом по сравнению даже с Docker. При нехватке ресурсов для настройки полномасштабного тестового кластера или необходимости выполнить комплексные тесты среднего уровня сложности мини-кластеры Yarn — достаточно неплохой вариант.

Мини-кластеры Yarn олицетворяют компромиссное решение, их можно настроить для работы в JVM. В ранних (до 1.6) версиях фреймворка Spark совмещать тесты на основе мини-кластеров Yarn с другими тестами было непросто из-за сохраняемого глобального состояния. В библиотеке spark-testing-base (https://github.com/holdenk/spark-testing-base) имеется пример использования мини-кластеров Yarn с фреймворком Spark.

Все эти варианты ограничиваются тестированием на одной машине, чего недостаточно для проверки возможности масштабирования системы. К счастью, существующие стандартные режимы развертывания Spark столь же хорошо подходят для комплексного тестирования.

Если у вас уже настроен кластер YARN, то выполнить комплексные тесты с помощью режима yarn-client не составит никаких проблем. Если же у вас нет постоянно работающего кластера, то можно с легкостью динамически запускать кластер по мере необходимости, задействуя сценарии EC2 фреймворка Spark. Режимы развертывания Spark подробно описаны в главе 7 Learning Spark, а также в разделе «Обзор кластерного режима» документации по фреймворку (http://spark.apache.org/docs/latest/cluster-overview.html).

Контроль производительности

Основная причина, по которой имеет смысл использовать Spark (и читать эту книгу), — высокоэффективная обработка больших данных. А потому важно убедиться, что производительность при вашем конкретном способе применения фреймворка соответствует ожидаемой.

Контроль производительности с помощью счетчиков Spark

Во время выполнения заданий Spark отслеживает значения множества счетчиков, немалая часть которых может пригодиться при отладке проблем с производительностью. Веб-интерфейс Spark предоставляет доступ ко многим счетчикам, а чтобы обращаться к ним программно, необходимо зарегистрировать объект `SparkListener` (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.scheduler.SparkListener) для сбора информации. Spark использует обратные вызовы для выдачи метрик, а большую часть нужной информации о производительности можно получить с помощью метода `onTaskEnd`, для чего фреймворк предоставляет класс `SparkListenerTaskEnd` (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.scheduler.SparkListenerTaskEnd). В числе этих метрик: количество прочитанных байтов, время выполнения, количество прочитанных записей и др. (пример 8.18).

Пример 8.18. Простой объект SparkListener для получения времени выполнения

```
class PerfListener extends SparkListener {
  var totalExecutorRunTime = 0L
  var jvmGCTime = 0L
  var recordsRead = 0L
  var recordsWritten = 0L
  var resultSerializationTime = 0L

  /**
   * Вызывается при завершении задачи
   */
  override def onTaskEnd(taskEnd: SparkListenerTaskEnd) {
    val info = taskEnd.taskInfo
    val metrics = taskEnd.taskMetrics
    updateMetricsForTask(metrics)
  }

  private def updateMetricsForTask(metrics: TaskMetrics): Unit = {
    totalExecutorRunTime += metrics.executorRunTime
    jvmGCTime += metrics.jvmGCTime
    resultSerializationTime += metrics.resultSerializationTime
    recordsRead += metrics.inputMetrics.recordsRead
    recordsWritten += metrics.outputMetrics.recordsWritten
  }
}
```

С помощью этого слушателя можно быстро проконтролировать общее время выполнения (пример 8.19).

Пример 8.19. Тест для получения времени выполнения

```
test("Производительность подсчета числа слов") {
  val listener = new PerfListener()
  sc.addSparkListener(listener)
  doWork(sc)
  println(listener)
  assert(listener.totalExecutorRunTime > 0)
  assert(listener.totalExecutorRunTime < 10000)
}
```

Проекты, предназначенные для контроля производительности

Существуют два проекта, часто применяемых при контроле производительности заданий Spark. Пакет spark-perf (https://github.com/databricks/spark-perf) от компании Databricks предназначен для сравнения производительности различных версий фреймворка Spark, но им можно воспользоваться и для тестирования производительности нашего собственного кода. Spark-perf написан на языке Python, но в основном служит для проверки производительности кода на языке Scala. В докладе Testing Spark: Best Practices («Тестирование Spark: рекомендуемые практики») (https://spark-summit.org/2014/wp-content/uploads/2014/06/Testing-Spark-Best-Practices-Anupama-Shetty-Neil-Marshall.pdf) на конференции Spark Summit — 2014 предлагается задействовать Gatling (https://gatling.io/) совместно с Spark Jobserver (https://github.com/spark-jobserver/spark-jobserver), что позволяет делать задания Spark видимыми как REST-сервисы для целей тестирования производительности.

Валидация заданий

Прежде чем использовать результаты выполнения заданий Spark, необходимо убедиться, что они соответствуют ожиданиям. Важную роль при этом играет валидация заданий. С помощью накопителей можно отслеживать значимую для задания информацию, например количество корректных и некорректных записей или пользователей, для которых у вас нет рекомендаций.

Для валидации заданий можно воспользоваться частью информации, о которой мы говорили в подразделе «Контроль производительности с помощью счетчиков Spark» на с. 243. Можно задействовать тот же слушатель с агрегированием результатов на всех этапах.

Накопители Spark также можно использовать для валидации заданий. Распространенный сценарий применения накопителей — отслеживание количества некорректных записей при обработке данных. Эти накопители можно не только задействовать для отладки, но и сделать их частью автоматизированного набора тестов для валидации. Поскольку накопители Spark страдают от проблемы периодического (причем иногда непредсказуемого) двойного учета прибавляемых к ним значений, то важно делать все правила валидации относительными (например, количество некорректных записей/всего записей).

Упростить написание этих правил призван проект spark-validator (https://github.com/holdenk/spark-validator), но на момент написания данной книги он находится в зачаточном состоянии. Относительные правила можно также создавать путем фиксации значений из предыдущих запусков теста. Так, количество прочитанных записей должно находиться в пределах определенной дисперсии от среднего значения по предыдущим заданиям (пример 8.20).

Пример 8.20. Валидация минимального процента корректных записей с помощью spark-validator

```
val validationRules = List[ValidationRule](
  new AbsolutePercentageRule(
    "некорректныеЗаписи", "корректныеЗаписи", Some(0.0), Some(1.0)))
val vc = new ValidationConf(tempPath, "job_7", true, validationRules)
val sqlCtx = new SQLContext(sc)
val validator = Validation(sqlCtx, vc)

val valid = sc.accumulator(0)
validator.registerAccumulator(valid, "корректныеЗаписи")

val invalid = sc.accumulator(0)
validator.registerAccumulator(invalid, "некорректныеЗаписи")

runTwoCounterJob(sc, valid, invalid)
```

Резюме

В этой главе мы рассмотрели создание тестов для проверки как функциональности, так и производительности нашего кода Spark. Мы охватили методики написания локальных тестов, а также сравнения наборов RDD и `DataFrame` в случае слишком больших объемов проверяемых данных. Тесты, которые вы создадите на основе этих методик, позволят вам чувствовать себя уверенно при рефакторинге программ, нацеленном на повышение производительности описанными в предыдущих главах способами.

9 Spark MLlib и ML

У фреймворка Spark есть две библиотеки машинного обучения: Spark MLlib и Spark ML — с совершенно разными API, но очень похожими алгоритмами. Эти библиотеки унаследовали многие из относящихся к производительности нюансы API наборов RDD и `Dataset`, на которых они основаны, но есть у них и свои особенности. MLlib — исторически первая из этих библиотек, перешедшая в фазу лишь поддержки/исправления ошибок. При обычных обстоятельствах мы бы на ней не останавливались, а сосредоточились на новом API, однако не все возможности существующих алгоритмов были перенесены в новый API Spark ML. Spark ML — более новая (вдохновленная примером scikit-learn) библиотека, находящаяся в фазе активной разработки.

Выбор между библиотеками Spark MLlib и Spark ML

На первый взгляд, самое очевидное различие между библиотеками MLlib и ML — используемые типы данных: первая поддерживает наборы RDD, а вторая — наборы `DataFrame` и `Dataset`. Различие в форматах данных не столь существенно, ведь обе библиотеки работают с наборами RDD/`Dataset` векторов, которые легко можно преобразовывать из формата RDD в `Dataset` и наоборот.

С точки зрения подхода к архитектуре основная цель Spark MLlib — предоставить базовый набор алгоритмов, оставляя пользователю решение львиной доли задач конвейеризации данных, их очистки, подготовки и выбора признаков. Spark ML же (вдохновленная примером библиотеки scikit-learn) старается предоставить пользователю API для всего, начиная от подготовки данных и заканчивая обучением модели.

В настоящее время единственный доступный вариант при необходимости выполнить потоковую обработку или динамическое обучение (online training) — использовать API библиотеки MLlib. Ряд алгоритмов из библиотеки Spark MLlib поддерживают обучение на потоковых данных с применением API `DStream` пакета Spark Streaming, которое мы рассмотрим в разделе «Организация потоковой обработки с помощью Spark» на с. 282. В библиотеке Spark ML на данный момент поддержка потоковой обработки отсутствует (см. SPARK-16424 (https://issues.apache.org/jira/browse/SPARK-16424)), над потоковыми `Dataset` ведутся активные работы, и пока сложно сказать, когда поддержка появится.

Выбирать между MLlib и ML желательно с прицелом на будущее. В библиотеке Spark ML и далее станут появляться новые возможности, которые не будут переноситься в библиотеку MLlib, находящуюся на этапе только исправления ошибок.

Интегрированный API конвейеров библиотеки Spark ML упрощает реализацию метаалгоритмов, таких как поиск по значениям параметров по различным компонентам. Оба API поддерживают алгоритмы регрессии, классификации и кластеризации. Если вы еще не выбрали библиотеку для вашего проекта, то имеет смысл по умолчанию выбирать Spark ML, ведь это основная активно развивающаяся библиотека машинного обучения для Spark.

Работаем с библиотекой MLlib

Многие факторы, касающиеся производительности при работе с Spark Core, применимы также и к MLlib. Один из самых очевидных — повторное использование наборов RDD, ведь во многих алгоритмах машинного обучения применяются итеративные вычисления или оптимизации, поэтому очень важно сохранять входные данные в нужных точках.

Алгоритмы обучения с учителем в API Spark MLlib ориентируются на наборы RDD маркированных точек, а алгоритмы без учителя используют наборы RDD векторов. Эти маркированные точки и векторы являются уникальными для библиотеки MLlib, они отличаются от классов векторов языка Scala и эквивалентных классов библиотеки Spark ML.

Знакомимся с библиотекой MLlib (устройство и импорты)

Добавить библиотеку MLlib в проект можно так же, как и другие компоненты Spark. Упростить эту задачу позволят шаги, описанные в подразделе «Управление зависимостями Spark» на с. 50. Координаты Maven для библиотеки MLlib фреймворка Spark версии 2.1: `org.apache.spark:spark-mllib_2.11:2.1.0`.

Импорты для библиотеки MLlib несколько разбросаны по сравнению с другими компонентами Spark (см. показанные в примере 9.1 импорты, используемые для обучения простой модели классификации).

Пример 9.1. Простые импорты MLlib для создания модели логистической регрессии

```
import com.github.fommil.netlib.BLAS.{getInstance => blas}
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.classification.{LogisticRegressionWithLBFGS,
  LogisticRegressionModel}
// Переименовываем Vector в SparkVector, чтобы избежать конфликтов
// с классом Vector языка Scala
import org.apache.spark.mllib.linalg.{Vector => SparkVector}
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.feature._
```

Отдельные алгоритмы структурированы по назначению, так что алгоритмы регрессии располагаются в `org.apache.spark.mllib.regression`, большинство алгоритмов классификации — в `org.apache.spark.mllib.classification`, а алгоритмы кластеризации — в `org.apache.spark.mllib.clustering`. Алгоритмы, связанные с деревьями (деревья принятия решений, случайные леса и т. п.), находятся отдельно в `org.apache.spark.mllib.tree`. Помимо обычных алгоритмов машинного обучения, пакет `org.apache.spark.mllib.feature` предоставляет ограниченный набор инструментов для подготовки данных.

Некоторые алгоритмы можно использовать как для классификации, так и для регрессии, поэтому один и тот же алгоритм может встречаться и в org.apache.spark.mllib.regression, и в org.apache.spark.mllib.classification с различными API.

Класс `LabeledPoint`, необходимый для всех алгоритмов машинного обучения с учителем в MLlib, находится в пакете `org.apache.spark.mllib.regression`. Требуемый для формирования `LabeledPoint` и используемый непосредственно алгоритмами кластеризации класс `Vector` — в `org.apache.spark.mllib.linalg`.

Во всех примерах в этом разделе мы будем ссылаться на класс Vector библиотеки MLlib как на SparkVector во избежание конфликтов с классом вектора языка Scala.

Название Vector фреймворка Spark легко может вступить в конфликт с другими пакетами, включая Scala. Это способно привести к появлению сбивающих с толку сообщений об ошибках.

Кодирование признаков в библиотеке MLlib и подготовка данных

Для выбора признаков и масштабирования необходимо нахождение данных во внутреннем формате Spark, так что, прежде чем говорить об этом, взглянем на кодирование данных в требуемый формат.

После кодирования во многих конвейерах данных машинного обучения происходит фильтрация данных, чтобы исключить некорректные или искаженные записи, которые могут привести к проблемам в любой итоговой модели. Пакет MLlib фреймворка Spark предполагает выполнение фильтрации с помощью преобразований наборов RDD.

Во многих случаях удачным вариантом для обнаружения аномальных значений и нормализации признаков будет использование каких-либо из тех методик работы с квантилями/аномалиями, которые мы исследовали при работе с задачей Златовласки в разделе «Пример со Златовлаской» на с. 151, хотя советуем рассмотреть применение вместо них приближенных алгоритмов.

После выполнения первоначальной фильтрации можно воспользоваться удобными инструментами для выбора признаков и масштабирования, предоставляемыми библиотекой MLlib. Эти преобразователи признаков подходят для выбора и кодирования признаков и масштабирования.

Вам не обязательно ограничиваться инструментами масштабирования признаков и кодирования библиотеки MLlib; вы можете (и, вероятно, должны) писать собственные.

Работа с векторами фреймворка Spark

Внутренний формат вектора Spark отличается от аналогичного формата языка Scala, и существуют отдельные векторные библиотеки для MLlib и ML. Вместо непосредственного формирования векторов фреймворк предоставляет объект-фабрику `org.apache.spark.mllib.linalg.Vector`, способную формировать как плотные, так и разреженные векторы. При наличии массива признаков можно непосредственно сформировать плотный вектор Spark с помощью метода `Vector.dense` (пример 9.2).

Пример 9.2. Создание плотного вектора Spark

```
def toSparkVectorDense(input: Array[Double]) = {
  Vectors.dense(input)
}
```

Если вы хотите представить имеющийся у вас плотный вектор в виде разреженного, то можно вызвать для него метод `toSparse` или непосредственно создать разреженный вектор, передав последовательность ненулевых кортежей `(индекс, значение)` в метод `Vector.sparse`.

Внимательно следите за тем, чтобы создавать векторы правильного типа. Здесь может помочь переименование импортов, подобно тому как было сделано в примере 9.1.

Подготовка текстовых данных

Не все признаки можно кодировать непосредственно, в случае текстовых данных необходимо преобразовать их в числовой формат. Для кодирования таких данных существует утилита `Word2Vec` и класс `HashingTF`. Последний реализует одну из простейших операций над признаками, не требующую никакого обучения и непосредственно применимую к данным. `HashingTF` оперирует с наборами RDD из `Iterable[String]`, поэтому вы можете разбить свои данные на лексемы любым удобным образом. Скажем, простой текст на английском языке можно кодировать так, как показано в примере 9.3.

Пример 9.3. Простой преобразователь HashingTF для набора RDD строк с сохранением исходной записи

```
def hashingTf(rdd: RDD[String]): RDD[SparkVector] = {
  val ht = new HashingTF()
  val tokenized = rdd.map(_.split(" ").toIterable)
  ht.transform(tokenized)
}
```

Хотя некоторые из утилит кодирования библиотеки MLlib возвращают объект типа SparkVector, фреймворк Spark требует передачи всех признаков в одном входном векторе (а не нескольких).

Несмотря на простоту вышеописанного подхода, при нем возвращается объект типа `SparkVector` и отбрасывается все, кроме результата преобразования `HashingTF`. Использовать это напрямую нежелательно, ведь, скорее всего, нужно будет сохранить смесь признаков и информации о метках. Вместо того чтобы применять функцию `transform` непосредственно к наборам RDD во время подготовки данных, можно задействовать ее для кодирования строковых записей в специальном ассоциативном массиве, как показано в примере 9.4.

Пример 9.4. Простой преобразователь HashingTF для набора RDD строк с сохранением исходной записи

```
def toVectorPerserving(rdd: RDD[RawPanda]): RDD[(RawPanda, SparkVector)] = {
  val ht = new HashingTF()
  rdd.map{panda =>
    val textField = panda.pt
    val tokenizedTextField = textField.split(" ").toIterable
    (panda, ht.transform(tokenizedTextField))
  }
}
```

HashingTF изменил алгоритм хеширования по умолчанию с функции хеширования языка Scala на MurmurHash3 в Spark 2.0, так что при обновлении существующего конвейера может потребоваться повторное обучение.

Для некоторых из преобразователей признаков, скажем `Word2Vec`, необходимо обучение, подобно тому как происходит в «традиционных» моделях машинного обучения. Основное отличие в том, что их результаты, скорее всего, нет смысла использовать непосредственно. Для обучения модели `Word2Vec` нужно разбить входные данные на лексемы, после чего просто вызвать метод `fit` экземпляра `Word2Vec`, который вернет объект `Word2VecModel`, как показано в примере 9.5.

Пример 9.5. Обучение модели Word2Vec

```
def word2vecTrain(rdd: RDD[String]): Word2VecModel = {
  // Разбиение данных на лексемы
  val tokenized = rdd.map(_.split(" ").toIterable)
  // Формирование модели Word2Vec
  val wv = new Word2Vec()
  wv.fit(tokenized)
}
```

Получившаяся в результате `Word2VecModel` несколько отличается от большинства преобразователей, порождаемых оценивателями. Большинство генерируемых оценивателями моделей выполняют преобразование или предсказание для одного и того же входного типа данных, обычно векторов. Вместо этого для обучения при модели `Word2VecModel` требуется поле для фраз, а итоговая модель преобразует поля для отдельных слов, как показано в примере 9.6.

Пример 9.6. Использование модели Word2Vec

```
def word2vec(sc: SparkContext, rdd: RDD[String]): RDD[SparkVector] = {
  // Разбиение данных на лексемы
  val tokenized = rdd.map(_.split(" ").toIterable)
  // Формирование модели Word2Vec
  val wv = new Word2Vec()
```

```
  val wvm = wv.fit(tokenized)
  val wvmb = sc.broadcast(wvm)
  // Теперь WVM способна преобразовывать отдельные слова.
  println(wvm.transform("панда"))
  // Длина вектора равна 100 — мы воспользуемся этим при создании
  // на основе WVM-преобразователя, работающего с фразами
  val vectorSize = 100
  // Функция преобразования работает со словами, но наши входные данные
  // представляют собой целые фразы
  tokenized.map{words =>
    // Если в words не содержится ничего, возвращаем пустой вектор
    if (words.isEmpty) {
      Vectors.sparse(vectorSize, Array.empty[Int], Array.empty[Double])
    } else {
      // Если в words содержатся какие-либо фразы, формируем текущую
      // сумму векторов для каждого слова
      val sum = Array[Double](vectorSize)
      words.foreach { word =>
        blas.daxpy(
          vectorSize, 1.0, wvmb.value.transform(word).toArray, 1, sum, 1)
      }
      // Умножаем ее на количество слов
      blas.dscal(sum.length, 1.0 / words.size, sum, 1)
      // И обертываем в вектор Spark
      Vectors.dense(sum)
    }
  }
}
```

Транслирование модели может существенно повысить производительность, особенно при больших и сложных моделях. В примере 9.6 мы транслируем модель, так что у каждого исполнителя будет только одна копия.

Подготовка данных для машинного обучения с учителем

Перед использованием алгоритмов для маркированных данных сначала нужно создать объект `LabeledPoint` с метками и вектор признаков. Для `LabeledPoint` необходимо, чтобы метки были числами типа `double` аналогично тому, что элементы вектора тоже должны быть такими числами. Как и с кодированием признаков, если метки числовые, то преобразовать типы будет несложно, однако для других типов придется задействовать пользовательскую функцию или аналогичную методику (допустим, `StringIndexer`).

Возвращаясь к нашему примеру со Златовлаской, видим, что у нас есть массив признаков панд, а также булев признак, указывающий, счастлива ли панда. Если считать, что эти данные уже более или менее очищены, то можно подготовить их для использования в MLlib так, как показано в примере 9.7.

Вместо кодирования метки в качестве признака с нулевым индексом, как в некоторых системах, в Spark и LabeledPoint используют отдельную метку.

Пример 9.7. Преобразование RawPanda в LabeledPoint

```
def toLabeledPointWithHashing(rdd: RDD[RawPanda]): RDD[LabeledPoint] = {
  val ht = new HashingTF()
  rdd.map{rp =>
    val hashingVec = ht.transform(rp.pt)
    val combined = hashingVec.toArray ++ rp.attributes
    LabeledPoint(booleanToDouble(rp.happy),
      Vectors.dense(combined))
  }
}
```

Один из простейших подходов на случай необходимости работы со строковыми метками — создание ассоциативного массива, как в примере 9.8.

Пример 9.8. Реализация поиска по метке

```
def createLabelLookup[T](rdd: RDD[T]): Map[T, Double] = {
  val distinctLabels: Array[T] = rdd.distinct().collect()
  distinctLabels.zipWithIndex
    .map{case (label, x) => (label, x.toDouble)}.toMap
}
```

Масштабирование и выбор признаков способны привести к существенному улучшению результатов некоторых алгоритмов и оптимизаторов, и в библиотеке MLlib для этого есть преобразователь признаков. Вы можете пропустить данный шаг, если торопитесь и просто создаете модельный образец, но мы советуем всегда тщательно продумывать используемые в модели признаки. Как и `Word2Vec`, их нужно обучить на входных данных, но, к счастью, полученные в результате модели можно применять непосредственно, без той сложной логики, которую мы задействовали для `Word2Vec` (пример 9.9).

Пример 9.9. Масштабирование признаков

```
// Обучает преобразователь масштаба признаков и возвращает его
// и отмасштабированные признаки
def trainScaler(rdd: RDD[SparkVector]): (StandardScalerModel, RDD[SparkVector]) = {
  val scaler = new StandardScaler()
  val scalerModel = scaler.fit(rdd)
  (scalerModel, scalerModel.transform(rdd))
}
```

Если после масштабирования признаков обнаружатся аномальные значения, то можно опять вернуться к этапу фильтрации.

Для выбора признаков можно обучить на необработанных признаках `PCA` или `ChiSqSelector`, которые возвращают как преобразователи, так и информацию о выбранных признаках (пример 9.10).

Пример 9.10. Выбор десяти наиболее существенных признаков с помощью класса ChiSqSelector

```
def selectTopTenFeatures(rdd: RDD[LabeledPoint]):
    (ChiSqSelectorModel, Array[Int], RDD[SparkVector]) = {
  val selector = new ChiSqSelector(10)
  val model = selector.fit(rdd)
  val topFeatures = model.selectedFeatures
  val vecs = rdd.map(_.features)
  (model, topFeatures, model.transform(vecs))
}
```

Обучение моделей библиотеки MLlib

После выбора и масштабирования признаков наступает время обучения модели. У большинства алгоритмов библиотеки MLlib есть метод `run`, который принимает на входе набор RDD объектов `LabeledPoint` (алгоритмы машинного обучения с учителем) или объектов `Vector` (алгоритмы машинного обучения без учителя) и возвращает модель.

У каждого из алгоритмов есть свои параметры, способные существенно влиять на производительность[1], так что не жалейте времени на просмотр документации по используемому алгоритму. Несмотря на множество преимуществ настройки, большинство алгоритмов будет работать без каких-либо особых установок, как показано в примере 9.11.

Пример 9.11. Обучение простой модели классификации для предсказания счастья панд

```
def trainModel(rdd: RDD[LabeledPoint]): LogisticRegressionModel = {
  val lr = new LogisticRegressionWithLBFGS()
  val lrModel = lr.run(rdd)
  lrModel
}
```

Вышеприведенная модель (пример 9.11) была обучена с помощью параметров, скажем количества кластеров по умолчанию. При промышленной эксплуатации MLlib, вероятно, понадобится задать значения каких-то параметров. В отличие от моделей из Spark ML, тут у каждой модели есть свои методы настройки, так что внимательно изучите документацию API Spark по используемым моделям. Можно легко модифицировать пример 9.11, включив в него свободный член регрессии, как в примере 9.12.

[1] Как время выполнения, так и точность/чувствительность.

Пример 9.12. Обучение простой модели классификации для предсказания счастья

```
def trainModelWithInterept(rdd: RDD[LabeledPoint]): LogisticRegressionModel = {
  val lr = new LogisticRegressionWithLBFGS()
  lr.setIntercept(true)
  val lrModel = lr.run(rdd)
  lrModel
}
```

Один из недостатков библиотеки MLlib — из-за отсутствия единого формата параметров в ней сложно реализовывать метаалгоритмы (например, поиск значений параметров), по сравнению с библиотекой Spark ML, в которой есть единый формат и утилиты для поиска значений параметров.

Предсказание

Теперь, когда уже есть модель, время задействовать ее для предсказания значений. Простейший способ — использовать в том же кластере Spark, где она обучалась, так как при этом модель не нужно будет экспортировать, а потом загружать. Однако во многих сценариях применения требуется меньшая задержка, чем может обеспечить механизм пакетной обработки Spark. Обычно не имеет смысла использовать кластер Spark для динамического предсказания, разве что в вашем случае микропакетный подход работает. Spark обеспечивает ограниченную поддержку предсказания по одной записи за раз, но для этого все равно нужен локальный кластер.

У большинства моделей MLlib есть функции `predict`, работающие с наборами RDD объектов типа SparkVector, а у некоторых — отдельная функция `predict` для обработки по одному вектору за раз. Это справедливо не для всех моделей; так, у `LDAModel` вместо функции `predict` есть метод `topicDistribution`, поэтому обязательно загляните в документацию API по используемой модели. Продолжая пример 9.11, можно задействовать API пакетного предсказания, чтобы предсказать, какие панды счастливы (пример 9.13). Опять же пакетный API подходит не для всех пакетных сценариев применения, поскольку не сохраняет исходную запись — вероятно, в пакетном режиме API для отдельных записей окажется даже удобнее (как в примере 9.4).

Пример 9.13. Предсказываем, счастливы ли панды

```
def predict(model: LogisticRegressionModel, rdd: RDD[SparkVector]): RDD[Double] = {
            model.predict(rdd)

   }
```

Некоторые модели при использовании в пакетном режиме будут пытаться транслировать свои внутренние данные ради снижения издержек. Как и в случае преобразований признаков, при необходимости задействовать отдельную функцию предсказания в преобразовании имеет смысл транслировать модель.

Выдача и сохранение

Сохранение модели и выдача тесно связаны, так как при многих схемах развертывания приходится задействовать для выдачи результатов набор машин, отличный от применяемого для обучения. Даже если вы используете модель только для пакетного предсказания по записям в заданиях Spark, все равно вы, вероятно, захотите сохранить модель, чтобы обратиться к ней в различных заданиях Spark.

Библиотека MLlib поддерживает экспорт в два формата: внутренний формат Spark и PMML (Predictive Model Markup Language, язык разметки для прогнозного моделирования (https://en.wikipedia.org/wiki/Predictive_Model_Markup_Language)). В библиотеке MLlib сохранение модели реализовано с помощью типажей `Saveable` и `PMMLExportable`. Первый предоставляет экспорт во внутренний формат Spark, легко читаемый в Spark и реализованный для множества моделей. Зачастую `Saveable` также оказывается более экономным в смысле расходуемого места. Объем выводимых данных при использовании `PMMLExportable` больше, чем при `Saveable`, и, как ни странно, модели, экспортированные с помощью экспорта PMML, нельзя загрузить обратно в Spark. Однако преимуществом экспорта PMML является возможность чтения во внешних системах.

Saveable (внутренний формат)

У типажа `Saveable` есть функция `save`, принимающая в качестве параметров объект `SparkContext` и целевой путь и записывающая модель во внутреннем формате Spark, как показано в примере 9.14. Внутренний формат состоит из метаданных в формате JSON (тип модели, количество признаков и классов) и «полупрозрачных» данных в формате Parquet, отражающих саму модель. Этот формат не является переносимым, но без проблем читается свежими версиями Spark.

Пример 9.14. Экспорт модели во внутренний формат фреймворка Spark

```
// Сохранение в каталог internal – удаленный путь
   model.save(sc, path + "/internal")
```

Для загрузки модели, сохраненной во внутреннем формате Spark, необходимо вызвать статический метод `load` соответствующего модели объекта. Использовать этот метод не очень просто, поскольку он не всегда присутствует в сгенерированной документации по API или в автодополнении вашего IDE. Загрузить обратно *выгруженную* в примере 9.14 модель можно так, как показано в примере 9.15.

Пример 9.15. Загрузка обратно экспортированной модели LogisticRegressionModel во внутреннем формате Spark

```
def load(sc: SparkContext, path: String): LogisticRegressionModel = {
  LogisticRegressionModel.load(sc, path + "/internal")
}
```

При необходимости работать вне Spark с моделью, которую нельзя экспортировать в формат PMML, но можно сохранить, будет уместно написать собственный код загрузки модели для разбора внутреннего формата Spark.

PMML

Типаж `PMMLExportable` обеспечивает стандартизированный формат вывода, хотя и реализован для меньшего подмножества моделей и не допускает непосредственной загрузки обратно в Spark (пример 9.16).

Пример 9.16. Экспорт модели в форматы PMML

```
// Сохранение в формат PMML — удаленный путь
model.toPMML(sc, path + "/pmml")
// Сохранение в формат PMML — локальный путь
model.toPMML(path + "/pmml")
```

Хотя Spark и поддерживает экспорт данных в PMML, но поскольку этот формат предназначен в основном для стыковки с внешними проектами, то загрузка экспортированных данных в формате PMML оставляется на откуп конечных пользователей.

Для загрузки и оценки экспортированных моделей PMML можно воспользоваться проектом JPMML, который, однако, лицензирован в соответствии с AGPL.

Пользовательские форматы

Не следует ограничиваться имеющимися в Spark встроенными форматами хранения, хотя для выхода за их пределы может понадобиться обращение к внутренним API Spark. Экспорт данных в пользовательские форматы требует доступа к внутренним API, мы обсудим его в контексте библиотеки Spark ML в разделе «Общие соображения о выдаче» на с. 278.

Некоторые особенно рисковые пользователи, не планировавшие задействовать свои модели с разными версиями Spark, успешно применяли сериализацию Java. Это очень ненадежный подход, способный прекращать работать после обновления Spark. Впрочем, использование данной сериализации позволяет быстро создать опытный образец моделей Spark MLlib, у которых отсутствует поддержка сохранения/загрузки.

Библиотека Spark MLlib вряд ли является удачным вариантом для выдачи с малой задержкой: в ее текущей реализации загрузка модели Spark, экспортированной

нативным образом, требует объекта SparkContext, а также всех обычных зависимостей Spark. Некоторые из разработчиков библиотеки Spark ML рассматривают возможность выделения кода выдачи результатов, но эти исследования еще далеки от завершения.

При наличии уже существующей, но не поддерживающей формат PMML системы выдачи можно применить другой подход: вручную расширить модель и написать пользовательский код экспорта. Это не очень надежный вариант, поскольку нюансы представления модели способны меняться от версии к версии, но может оказаться оптимальным, если нужна всего лишь поддержка ограниченного числа моделей и система выдачи результатов уже есть.

Оценка модели

Помимо возможностей, связанных с выполнением предсказаний, во многих моделях также содержится информация о точности и других сводных статистических показателях, которые могут быть интересны исследователям данных (или разработчикам), поэтому обращайте внимание на все поля полученной модели. Так, модель k-средних (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.clustering.KMeans) дает доступ к центрам кластеров, а модель логистической регрессии (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.mllib.classification.LogisticRegressionModel), которую мы обучали для предсказания счастливых панд в примере 9.12, содержит свободный член уравнения регрессии и веса признаков.

Помимо встроенных сводных показателей моделей, пакет org.apache.spark.mllib.evaluation содержит средства вычисления нескольких различных показателей по заданным предсказаниям и контрольным данным (ground truth). Объект MLUtils предоставляет функцию kfold, сегментирующую данные для оценки.

Работаем с библиотекой Spark ML

В основе API библиотеки Spark ML лежит понятие конвейера, состоящего из различных этапов. Каждый из них выполняет свою задачу, причем существуют этапы для задач, начиная от очистки данных до выбора признаков с помощью применения алгоритма машинного обучения. Для сталкивавшихся с библиотекой scikit-learn большая часть архитектуры покажется знакомой. Этапы конвейера сгруппированы в оцениватели и преобразователи.

Оцениватели, такие как NaiveBayes, требуют обучения перед использованием, в то время как преобразователи, например алгоритм кодирования векторов, можно применять непосредственно. API конвейеров упрощает работу со множеством утилит подготовки и очистки данных, помимо обычных алгоритмов машинного обучения, а также позволяет сохранять весь конвейер для дальнейшего использования.

Устройство и импорты библиотеки Spark ML

Как Spark ML, так и Spark MLlib в настоящее время располагаются в одном компоненте фреймворка Spark, Maven-координаты которого в Spark 2.0: `org.apache.spark:spark-mllib_2.11:2.0.0`. После добавления его в сборку можно приступить к импорту частей библиотеки Spark ML, нужных для конкретной задачи машинного обучения.

Утилиты, предназначенные для формирования конвейера машинного обучения, располагаются в корне пакета `org.apache.spark.ml`. Для каждого из различных семейств алгоритмов (классификация, регрессия, рекомендация и кластеризация) существует отдельный пакет в пакете ML (например, алгоритмы классификации располагаются в пакете `org.apache.spark.ml.classification`). Помимо обычных алгоритмов машинного обучения, имеется и обширная коллекция преобразователей признаков в пакете `org.apache.spark.ml.feature` и метаалгоритмов в пакете `org.apache.spark.ml.tuning`.

Стандартные импорты показаны в примере 9.17, хотя, вероятно, вы захотите избежать использования джокерных символов. Помимо этих стандартных импортов, не помешает задействовать переименование, как показано в примере 9.18.

Пример 9.17. Стандартные импорты библиотеки Spark ML

```
import org.apache.spark.ml._
import org.apache.spark.ml.feature._
import org.apache.spark.ml.classification._
```

Как и Spark MLlib, библиотека Spark ML содержит компоненты для базовых операций линейной алгебры в пакете `org.apache.spark.ml.linalg`, а начиная со Spark 2.0, пакет линейной алгебры доступен также в виде целевого JAR без зависимостей от остального Spark. Этот пакет служит для упрощения локальной выдачи результатов.

В примерах из этого раздела мы будем ссылаться на класс Vector фреймворка Spark как SparkVector, чтобы избежать конфликтов с классом векторов языка Scala.

Пример 9.18. Переименованный импорт для SparkVector

```
import org.apache.spark.ml.linalg.{Vector => SparkVector}
```

Следует отметить, что пакет org.apache.spark.mllib.Vector отличается от пакета org.apache.spark.ml.Vector.

Можно воспользоваться функцией fromML класса org.apache.spark.mllib.linalg.Vectors для преобразования из формата ml в формат mllib.

Этапы конвейера

Этапы конвейера — основные «строительные блоки» библиотеки Spark ML. Как задачи подготовки данных, так и классическое обучение модели существуют в виде этапа. Такие этапы могут представлять собой или *преобразователи* (transformers), не требующие обучения на данных перед использованием, или *оцениватели* (estimators), которые необходимо обучить, прежде чем применять.

Хотя обычно этапы рассматриваются в контексте полного конвейера, который мы рассмотрим далее, может оказаться удобно выполнять их отладку по одному. При использовании в конвейере различия между преобразователями и оценивателями незаметны, однако в случае применения вне конвейера отличия API становятся видны. Прежде всего при работе с преобразователями и оценивателями вне конвейера можно лучше разобраться в том, чем они отличаются и как работают.

Преобразователь — простейший из этапов конвейера, а оцениватель в результате подгонки возвращает преобразователь. Для работы этапов-преобразователей конвейера можно напрямую вызвать метод `transform`, принимающий в качестве входных данных объект `DataFrame`. Метод `transform` возвращает новый объект `DataFrame` с результатом преобразования, который обычно получается просто путем добавления (например, новых столбцов, полученных из преобразованных старых). Что же касается оценивателей, подогнать оцениватель на конкретном входном объекте `DataFrame` можно с помощью вызова метода `fit`. Последний возвращает этап-преобразователь конвейера, для которого затем можно вызвать метод `transform`.

Практически у всех этапов конвейера есть какие-либо базовые параметры, которые необходимо задать, чтобы использовать данные этапы (например, столбец объекта `DataFrame`, к которому необходимо применить преобразование). Поэтому прежде чем двигаться дальше, придется разобраться, как работают эти параметры.

Способ установки параметров одинаков у всех этапов конвейера, что позволяет создать метаалгоритмы, например, для поиска значений параметров. Данные параметры собраны вместе в разделе Parameters (Параметры) Javadoc каждого этапа, а геттеры и сеттеры — в разделах Parameter getters (Геттеры для параметров) и Parameter setters (Сеттеры для параметров) соответственно. Для начала вы можете задать настройки своих алгоритмов машинного обучения с помощью сеттеров для параметров, а геттеры пригодятся позднее. У некоторых параметров имеются значения по умолчанию, хотя они зачастую были выбраны из соображений обратной совместимости, а не максимальной производительности.

Хотя параметры по умолчанию часто выбираются из соображений обратной совместимости, они иногда меняются от версии к версии — так что ради повышения воспроизводимости результатов лучше указывать явным образом даже используемые значения по умолчанию.

Пользователи языка Python, работающие с библиотекой Spark ML, могут обнаружить, что документации относительно значений по умолчанию различных параметров нет. В подобных случаях рекомендуем заглянуть в версию данной документации для языка Scala. Однако между языками могут существовать различия, поэтому рекомендуемой практикой все равно является установка значений явным образом равными значениям по умолчанию.

Функция ExplainParams

При работе в командной оболочке можно воспользоваться функцией `ExplainParams()` для вывода всех параметров этапа, его документации и текущего значения. Для вывода отдельного параметра служит синтаксис `ExplainParams("paramName")`. В примере 9.19 показан образец этапа `Binarizer`[1] конвейера без перекрытия каких-либо параметров.

Пример 9.19. Результаты выполнения функции ExplainParams() на этапе Binarizer конвейера

```
inputCol: input column name (undefined)
outputCol: output column name (default: binarizer_8b03ca79966b__output)
threshold: threshold used to binarize continuous features (default: 0.0)
```

В большинстве этапов производится дополнительная валидация задаваемых значений (например, проверка того, что коэффициент масштабирования не равен нулю) при преобразовании схемы входного объекта DataFrame (скажем, выполнения для этого DataFrame операции transform или fit).

Как вы можете видеть, в примере 9.19 исходный столбец не задан и значение по умолчанию отсутствует. Следовательно, для использования данного этапа необходимо сначала задать их значения. Для этого можно применить функцию `setParameterName([значение])` (в данном случае `setInputCol("inputCol")`). Сеттеры обновляют конвейер, для которого вызваны, а также возвращают объект этапа ради упрощения организации цепочкой нескольких операций-сеттеров. Важно отметить, что сеттеры не копируют, а модифицируют объект этапа, для которого вызваны, и возвращают его же.

Во время подгонки/обучения преобразователя/модели значения параметров копируются из оценивателя в преобразователь[2].

[1] Преобразование в двоичную форму. — *Примеч. пер.*

[2] В Python этого сейчас не происходит, но для исправления подобной ситуации было зарегистрировано сообщение об ошибке.

Зачастую можно менять параметры и в модели: так, если вы используете один столбец для обучения, а другой — при «динамической» выдаче результатов, можете поменять входной столбец итоговой модели с помощью тех же сеттеров, что применяются к оценивателю. Будьте осторожны при изменении параметров модели, поскольку иногда это возможно, но на самом деле не оказывает никакого влияния (например, смена регуляризации).

Теперь, разобравшись с основами настройки этапов конвейера, можно перейти к использованию инструментов подготовки данных библиотеки Spark ML.

Кодирование данных

Большинство моделей Spark ML, хотя и получают в качестве параметра `Dataset[_]`, требуют кодирования данных в определенный формат. Признаки должны быть представлены столбцом типа `Vector`. При обучении с помощью алгоритма машинного обучения с учителем метки должны быть типа `Double`, а признаки — отдельным объектом `Vector`.

Некоторые алгоритмы в Spark предполагают, что столбец меток индексируется начиная с нуля, и могут вести себя неоптимальным образом (в смысле времени обучения или качества предсказаний), если входной столбец меток индексирован не с нуля. В решении данной проблемы может помочь преобразователь признаков StringIndexer, о котором мы поговорим ниже.

К счастью, в библиотеке Spark ML есть множество компонентов конвейера для подготовки данных. Они располагаются в пакете `org.apache.spark.ml.feature` и на момент написания данной книги включают более 35 алгоритмов, охватывающих все, начиная от `Binarizer` и `PCA` и заканчивая `Word2Vec`. Этапы, предназначенные для подготовки данных, могут быть как оценивателями (например, `Word2Vec` или `StringIndexer`), так и преобразователями (скажем, `HashingTF`).

Наиболее часто используемый преобразователь признаков — `VectorAssembler`; он позволяет преобразовать входные данные в формат, с которым могут работать модели машинного обучения из библиотеки Spark ML. Если данные уже представлены в виде числовых типов, то для применения `VectorAssembler` достаточно задать входные и требуемые выходные столбцы для сформированного вектора, как показано в примере 9.20.

Пример 9.20. Простой формирователь векторов

```
val assembler = new VectorAssembler()
assembler.setInputCols(Array("size", "zipcode"))
```

На случай текстовых входных данных в конвейере Spark ML имеются простейшие функции кодирования текста, например `Word2Vec`, `StopWordsRemover`, `NGram`, `IDF`, `HashingTF` и простой `Tokenizer`. Две последние функции, задействованные совместно, позволяют преобразовать входной текстовый столбец в числовой признак, подходящий для использования в Spark. В примере 9.21 мы демонстрируем, как это сделать вручную, но применение конвейера (которое мы рассмотрим в подразделе «Собираем все воедино в конвейер» на с. 266) значительно упрощает данный процесс.

Пример 9.21. Создаем лексический анализатор и используем результаты его работы для операции HashingTF

```
val tokenizer = new Tokenizer()
tokenizer.setInputCol("name")
tokenizer.setOutputCol("tokenized_name")
val tokenizedData = tokenizer.transform(df)
val hashingTF = new HashingTF()
hashingTF.setInputCol("tokenized_name")
hashingTF.setOutputCol("name_tf")
hashingTF.transform(tokenizedData)
```

Если в Spark понадобятся более мощные инструменты NLP[1], то загляните на сайт spark-packages или рассмотрите возможность расширения конвейера Spark своим кодом, как описано в подразделе «Расширение конвейеров Spark ML собственными алгоритмами» на с. 270.

Получив предназначенный для обучения объект `Vector` с признаками, можно заняться кодированием меток. Как понятно из названия[2], метод `StringIndexer` подходит для преобразования строковых значений в индексы — распространенный способ работы с данными, маркированными текстовыми метками. Данный метод также подходит для категориальных числовых типов входных данных, он удобен при работе с категориальными столбцами меток, которые могут быть индексированы не с нуля. `StringIndexer` представляет собой оцениватель, так как его цель — установить меткам, встречающимся чаще всего, минимальное значение индекса, при котором не будет никаких конфликтов. Чтобы отсортировать строковые значения по частоте появления, методу `StringIndexer` необходимо знать, какие строковые значения встречаются чаще всего. В результате подгонки индексатор строковых значений возвращает объект `StringIndexerModel` (как показано в примере 9.22), который затем можно использовать для преобразования данных.

---

[1] От англ. Natural Language Processing — обработка написанных на естественных языках текстов. — *Примеч. пер.*

[2] "string indexer " = "индексатор строковых значений". — *Примеч. пер.*

Пример 9.22. Обработка строковых меток с помощью метода StringIndexer

```
// Создаем простой индексатор строковых значений
val sb = new StringIndexer()
sb.setInputCol("name")
sb.setOutputCol("indexed_name")
// Создаем модель на основе входных данных
val sbModel = sb.fit(df)
```

Существует и обратный объекту `StringIndexerModel` преобразователь под названием `IndexToString`, используемый для преобразования предсказаний обратно в исходные метки. Теоретически он может работать на основе метаданных, закодированных в схеме объектов `DataFrame` (как в примере 9.23), но многие оцениватели не копируют метаданные из столбца меток в столбец предсказаний.

Пример 9.23. Для использования IndexToString необходимы метаданные

```
// Формируем обратную модель для преобразования из индексов
// в строковые значения после предсказания
val sbInverseMD = new IndexToString()
sbInverseMD.setInputCol("prediction")
```

Если в выходном столбце отсутствуют метаданные, то для восстановления исходных меток придется сформировать объект `IndexToString` вручную, как показано в примере 9.24.

Пример 9.24. Использование IndexToString без метаданных

```
// Формируем обратную модель для преобразования из индексов
// в строковые значения после предсказания
val sbInverse = new IndexToString()
sbInverse.setInputCol("prediction")
sbInverse.setLabels(sbModel.labels)
```

В случае обнаружения новых значений, скажем, вследствие того что одна из меток при k-блочном обучении отсутствовала, модель StringIndexerModel может или сгенерировать исключение, или пропустить запись. Выбрать, как она должна себя вести, можно с помощью параметра handleInvalid (значение его по умолчанию соответствует генерации исключения).

Очистка данных

Помимо простого преобразования данных в подходящий для работы Spark формат, дополнительное проектирование признаков способно весьма благотворно повлиять на производительность моделей. Преобразование `Binarizer` можно использовать для установки порогового значения конкретного признака, а `PCA` — для понижения размерности данных. Если обучаемые модели лучше работают с нормализованными данными, то можно задействовать преобразование `Normalizer` (по всему вектору признаков), как показано в примере 9.25, или `MinMaxScaler`

(по отдельному столбцу) для добавления этапа нормализации признаков в существующий конвейер.

Пример 9.25. Создание нормализатора

```
val normalizer = new Normalizer()
normalizer.setInputCol("features")
normalizer.setOutputCol("normalized_features")
```

Теперь, после первого шага подготовки данных для обучения, можно перейти к обсуждению алгоритмов машинного обучения библиотеки Spark ML.

Модели библиотеки Spark ML

В Spark есть множество алгоритмов машинного обучения, от классификации и регрессии до кластеризации. У каждой из моделей есть параметры настройки, описанные в документации Scala/Javadoc. У каждого из алгоритмов машинного обучения с учителем есть как минимум параметры `labelCol`, `featuresCol` и `predictionCol`.

Различные алгоритмы машинного обучения сгруппированы по назначению. А раз все семейства алгоритмов машинного обучения (классификация (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.classification.package), регрессия (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.regression.package), рекомендация (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.recommendation.package) и кластеризация (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.clustering.package)) сгруппированы по пакетам, для просмотра доступных алгоритмов достаточно заглянуть в соответствующий пакет. Помимо поставляемых непосредственно с фреймворком Spark моделей, можно добавить в конвейер Spark собственный код (см. подраздел «Расширение конвейеров Spark ML собственными алгоритмами» на с. 270) или применить алгоритмы сообщества разработчиков (см. раздел «Использование пакетов и библиотек, созданных сообществом разработчиков» на с. 297).

Создание и конфигурация этапа машинного обучения аналогичны конфигурации других оценивателей, таких как `StringIndexer`. У оценивателей для машинного обучения зачастую доступно больше параметров для настройки, чем у предназначенных для подготовки данных оценивателей, и подгонка первых занимает больше времени. У большинства из этих параметров имеются значения по умолчанию, так что создание оценивателя для машинного обучения не требует большого количества настроек, как показано в примере 9.26.

Пример 9.26. Создание этапа для наивного классификатора Байеса с минимальными настройками

```
val nb = new NaiveBayes()
nb.setLabelCol("happy")
nb.setFeaturesCol("features")
nb.setPredictionCol("prediction")
val nbModel = nb.fit(df)
```

При работе со Spark из командной оболочки или блокнота к этапам ML применимы те же приемы, которые мы обсуждали в подразделе «Функция ExplainParams» на с. 261.

Как и для других оценивателей, можно непосредственно вызывать метод `fit`, но использовать этап машинного обучения в конвейере гораздо удобнее, чем организовывать этапы цепочкой вручную, с помощью своего кода.

Собираем все воедино в конвейер

Интерфейс конвейеров библиотеки Spark ML позволяет объединить этапы подготовки данных и обучения модели. Такая возможность особенно удобна для сохранения конвейера или запуска метаалгоритмов (например, поиска значений параметров) по нескольким компонентам конвейера. Даже если не учитывать выгоды от упрощения сохранения или метаалгоритмов, организация преобразователей и оценивателей цепочкой в единообразный конвейер способна упростить код обучения модели по сравнению со связыванием вызовов методов `fit` и `transform` вручную.

После того как мы разобрались с созданием отдельных этапов конвейера, не составляет труда скомпоновать их в конвейер. Для конфигурации этапов и самих конвейеров используются одни и те же механизмы параметров, и конвейер тоже является оценивателем (при желании можно создать конвейер конвейеров, но особого смысла в данном действии нет).

Все, что требуется для построения конвейера, — создать экземпляр класса `org.apache.spark.ml.Pipeline` и вызвать метод `setStages`, передав ему массив этапов конвейера. Этапы из предыдущих примеров можно составить в единый конвейер, сэкономив усилия на преобразовании объекта `DataFrame` вручную, как показано в примере 9.27.

Пример 9.27. Формируем простой конвейер

```
val tokenizer = new Tokenizer()
tokenizer.setInputCol("name")
tokenizer.setOutputCol("tokenized_name")
val hashingTF = new HashingTF()
hashingTF.setInputCol("tokenized_name")
hashingTF.setOutputCol("name_tf")
val assembler = new VectorAssembler()
assembler.setInputCols(Array("size", "zipcode", "name_tf",
  "attributes"))
val nb = new NaiveBayes()
nb.setLabelCol("happy")
nb.setFeaturesCol("features")
nb.setPredictionCol("prediction")
val pipeline = new Pipeline()
pipeline.setStages(Array(tokenizer, hashingTF, assembler, nb))
```

Конвейеры удобны тем, что позволяют легко добавить в себя новые этапы, скажем, можно добавить в пример 9.27 нормализатор, как показано в примере 9.28.

Пример 9.28. Конвейер с нормализатором

```
val normalizer = new Normalizer()
normalizer.setInputCol("features")
normalizer.setOutputCol("normalized_features")
nb.setFeaturesCol("normalized_features")
pipeline.setStages(Array(tokenizer, hashingTF, assembler, normalizer, nb))
val normalizedPipelineModel = pipelineModel.transform(df)
```

Обучение конвейера

Раз конвейер — просто особая разновидность оценивателя, способная включать другие оцениватели, его можно использовать аналогичным образом. Вызов метода `fit()` с заданным объектом `Dataset` приведет к подгонке этапов конвейера по очереди (за исключением преобразователей) и возврату конвейера, состоящего из полностью обученных преобразователей, которые можно задействовать для предсказания.

Обращение к отдельным этапам

Возможно, понадобится обратиться к отдельному этапу конвейера, скажем, для получения отладочной информации или сохранения одного из компонентов вручную. Корневой класс `Pipeline` содержит параметр `stages`, представляющий собой массив этапов конвейера. После обучения итоговый объект `PipelineModel` содержит массив `stages`, состоящий из всех этапов конвейера после обучения. Обращение к этапам в языке Scala обычно требует приведения их к соответствующему типу явным образом, как показано в примере 9.29.

Пример 9.29. Обращение к отдельным этапам конвейера ML

```
val tokenizer2 = pipelineModel.stages(0).asInstanceOf[Tokenizer]
val nbFit = pipelineModel.stages.last.asInstanceOf[NaiveBayesModel]
```

Чаще всего нужно будет обращаться к последнему этапу конвейера для получения информации о производительности модели. Еще одна вероятная причина, по которой может понадобиться обращаться к отдельным этапам, — создание обратной модели для `StringIndexerModel` на основе ее меток для формирования этапа `IndexToString`.

Сохранение данных и библиотека Spark ML

Это может показаться парадоксальным, но при работе с алгоритмами машинного обучения Spark можно добиться большего быстродействия, если не кэшировать явным образом данные. В случае данных, не используемых повторно вне алгоритма

машинного обучения, многие итеративные алгоритмы сами заботятся о кэшировании или позволяют настраивать уровень сохранения с помощью свойства `intermediateStorageLevel`. Одним из первых шагов для каждой модели часто является преобразование входных данных в более удобный формат, и любой вид внутреннего сохранения оперирует этим внутренним форматом. По сути, модели сохраняют свои данные более эффективно — если, конечно, этот этап конвейера вообще может обеспечить сохранение.

Далеко не все этапы конвейеров поддерживают разумное сохранение. Если параметра сохранения нет и в Scaladoc не упоминается сохраняемость, то стоит заглянуть в пользовательский интерфейс Spark и посмотреть, не лучше ли будет сохранять входные данные конвейера вручную.

Вне зависимости от того, кэшируете ли вы входные данные явным образом или делегируете задачу кэширования подготовленных данных алгоритму, применимы все соображения по поводу выбора уровня хранения, приведенные в подразделе «Виды повторного использования: кэш, сохранение, контрольная точка, перетасовочные файлы» на с. 140.

Автоматизированный выбор модели (поиск значений параметров). Настройка модели представляет собой непростую задачу, особенно для специалистов, мало знакомых со сферой машинного обучения. К счастью, стандартизированный подход к параметрам моделей библиотеки Spark ML облегчает поиск, позволяя задавать параметры, по которым необходимо выполнить поиск, а также метод их оценки.

Если вы не имеете опыта работы с машинным обучением или поиска значений параметров, то не забудьте, что при автоматизированном выборе модели (или любой ручной настройке) необходимы отдельные обучающая, контрольная и проверочная последовательности данных. Это поможет точнее оценить эффективность модели.

Фреймворк Spark способен выполнить поиск по значениям различных параметров для автоматического выбора оптимальных параметров, а следовательно, и модели. Воспользуемся нашим примером конвейера 9.27 и сформируем координатную сетку для поиска оптимального коэффициента сглаживания, как показано в примере 9.30.

Пример 9.30. Простой поиск значения отдельного параметра

```
// Объект ParamGridBuilder формирует массив сочетаний значений параметров
val paramGrid: Array[ParamMap] = new ParamGridBuilder()
  .addGrid(nb.smoothing, Array(0.1, 0.5, 1.0, 2.0))
  .build()
```

Хотя обучение и оценка для каждой заданной конфигурации выполняются параллельно, поиск по различным конфигурациям происходит последовательно, причем для сужения заданного пространства поиска не используются никакие оптимизации. Поэтому требуемый объем времени может быстро расти по мере увеличения количества имеющихся вариантов.

В настоящее время фреймворк Spark предоставляет инструменты разбиения данных для оценки модели: разбиение для перекрестной проверки и разбиение для обучения/проверки. Оба автоматически кэшируют обучающую и проверочную последовательности на уровне хранилища в оперативной памяти, так что если ваша входная обучающая последовательность слишком велика для кэширования в оперативной памяти на рабочих узлах, то имеет смысл выполнить выборку из нее для поиска значений параметров.

Помимо выбора способа проведения испытаний, можно настроить метрику, применяемую для выбора оптимальной модели. В нашем текущем примере входные данные предназначены для бинарной классификации, так что в примере 9.31 используется оцениватель `BinaryClassificationEvaluator`, а для других типов данных существуют `RegressionEvaluator` и `MulticlassClassificationEvaluator`.

Пример 9.31. Поиск значений параметров с перекрестной проверкой

```
val cv = new CrossValidator()
  .setEstimator(pipeline)
  .setEstimatorParamMaps(paramGrid)
val cvModel = cv.fit(df)
val bestModel = cvModel.bestModel
```

Аналогично этапам конвейера можно настраивать алгоритм оценки. Чаще всего меняют оцениваемую метрику, доступную во всех встроенных алгоритмах оценки с помощью параметра `metricName`. `BinaryClassificationEvaluator` поддерживает метрики `areaUnderROC` и `areaUnderPR`. Другие алгоритмы оценки также поддерживают несколько различных метрик, которые можно найти в их Java/Scala: документации.

Можно реализовать и собственный алгоритм оценки путем наследования класса org.apache.spark.ml.evaluation.Evaluator.

Искомые параметры не обязательно ограничивать пределами одного этапа. Чтобы найти значения параметров для нескольких этапов одного конвейера, достаточно добавить дополнительные параметры в объект `ParamGridBuilder`. Возвращаясь к нашему примеру, можно легко организовать поиск значений параметров `numfeatures` и `binary` преобразователя `hashingTF` и параметра `p`-нормы нормализатора, как показано в примере 9.32.

Пример 9.32. Поиск значений параметров по всем этапам

```
val complexParamGrid: Array[ParamMap] = new ParamGridBuilder()
  .addGrid(nb.smoothing, Array(0.1, 0.5, 1.0, 2.0))
  .addGrid(hashingTF.numFeatures, Array(1 << 18, 1 << 20))
  .addGrid(hashingTF.binary, Array(true, false))
  .addGrid(normalizer.p, Array(1.0, 1.5, 2.0))
  .build()
```

Один из недостатков подобных громоздких поисков значений параметров — быстрый рост множества перебираемых моделей с каждым параметром, добавляемым в пространство поиска. Предпринимаются попытки организовать «интеллектуальный» поиск значений параметров в scikit-learn и подобных системах, но пока еще не во фреймворке Spark.

Ранние версии RegressionEvaluator иногда возвращали некорректные результаты. Если был возвращен NaN в качестве результата, то загляните в SPARK-14489 (https://issues.apache.org/jira/browse/SPARK-14489).

Эти инструменты оценки подходят для определения эффективности модели не только при поиске значений параметров, что может оказаться особенно полезно в случае необходимости контролировать корректность новых моделей перед использованием результатов в промышленной эксплуатации.

Расширение конвейеров Spark ML собственными алгоритмами

Хотя для конвейеров Spark ML существует множество алгоритмов, может понадобиться дополнительная функциональность в рамках модели конвейера. В библиотеке Spark MLlib это не вызывает проблем: можно вручную реализовать свой алгоритм с преобразованиями наборов RDD посередине использования встроенных функций. То же самое можно проделать и в случае конвейеров Spark ML, но это приведет к утрате некоторых приятных возможностей конвейеров, в частности возможности автоматически выполнять метаалгоритмы, такие как поиск значений параметров с перекрестной проверкой.

Чтобы добавить свой алгоритм в конвейер Spark, необходимо создать или объект `Estimator`, или объект `Transformer`, которые воплощают интерфейс `PipelineStage`. Интерфейс `Transformer` используется для алгоритмов, не требующих обучения, интерфейс `Estimator` — для алгоритмов, требующих этого. Оба интерфейса располагаются в пакете `org.apache.spark.ml` (и реализуют базовый типаж `PipelineStage`). Обратите внимание: обучение требуется не только для сложных моделей машинного обучения; даже для преобразования `MinMaxScaler` оно необходимо, чтобы определить диапазон, так как для этого ему нужно сначала просмотреть входные

данные. Если для вашего алгоритма требуется обучение, то нужно сформировать его в виде объекта `Estimator`, а не `Transformer`.

Непосредственно использовать PipelineStage нельзя, поскольку подгонка конвейера зависит от рефлексии, подразумевающей, что любой этап конвейера представляет собой или Estimator, или Transformer.

Пользовательские преобразователи

Даже если вы реализуете оцениватель, не помешает сначала разобраться, как создаются преобразователи, поскольку результатом работы оценивателя должен быть именно он.

Помимо самих собой разумеющихся функций `transform` или `fit`, все этапы конвейера должны содержать функцию `transformSchema` и конструктор копирования `copy` либо реализовывать класс, который бы их предоставлял. Конструктор `copy` используется для создания копии текущего этапа с включением всех только что заданных параметров и обычно просто вызывает метод `defaultCopy`, если у вашего класса нет каких-либо особенностей в смысле конструктора.

Конструктор этапа конвейера, а также делегирование задачи копирования показаны в примере 9.33.

Пример 9.33. Начало создания пользовательского этапа конвейера

```
class HardCodedWordCountStage(override val uid: String) extends Transformer {
  def this() = this(Identifiable.randomUID("hardcodedwordcount"))

  def copy(extra: ParamMap): HardCodedWordCountStage = {
    defaultCopy(extra)
  }
```

Функция `transformSchema` должна формировать нужный результат этапа конвейера на основе заданного набора параметров и исходной схемы. Большинство этапов конвейера просто добавляют новые поля, и лишь очень немногие из них удаляют уже существующие поля.

Если из-за размера записи в вашем конвейере возникают проблемы, то можно создать пользовательский этап конвейера для удаления лишних полей.

Помимо формирования выводной схемы, функция `transformSchema` должна проверить, что входная схема подходит для данного этапа (скажем, соответствует ли тип входного столбца ожидаемому). Именно в ней следует также выполнить валидацию параметров этапа. В примере 9.34 показана простая функция `transformSchema`,

предназначенная для строковых входных данных и векторных выходных, с жестко «зашитыми» входными и выходными столбцами.

Пример 9.34. Простая функция transformSchema

```
override def transformSchema(schema: StructType): StructType = {
  // Проверяем, что тип входных данных – строка
  val idx = schema.fieldIndex("happy_pandas")
  val field = schema.fields(idx)
  if (field.dataType != StringType) {
    throw new Exception(
      s"Input type ${field.dataType} did not match input type StringType")
  }
  // Добавляем возвращаемое поле
  schema.add(StructField("happy_panda_counts", IntegerType, false))
}
```

Реализовать алгоритмы, не требующие обучения или подгонки к входным данным, можно без особых проблем с помощью интерфейса `Transformer`, как мы начали делать в примере 9.33. Начать стоит с простого преобразователя — простейшего этапа конвейера, подсчитывающего количество слов во входном столбце, как показано в примере 9.35.

Пример 9.35. Подсчет количества слов во входном столбце

```
def transform(df: Dataset[_]): DataFrame = {
  val wordcount = udf { in: String => in.split(" ").size }
  df.select(col("*"),
    wordcount(df.col("happy_pandas")).as("happy_panda_counts"))
}
```

Для получения максимума отдачи от интерфейса конвейера желательно, чтобы этапы конвейера можно было настраивать с помощью интерфейса `params`. Эти общедоступные, но часто используемые в этапах Spark ML параметры по умолчанию, к сожалению, приватные, и приходится их переопределять. Параметры не только позволяют пользователям задавать значения, но и могут содержать простую логику проверки данных (так, параметр регуляризации должен быть неотрицательным значением). Два чаще всего применяемых параметра: входной и выходной столбцы, добавление которых в модель не представляет трудностей.

Можно использовать не только строковые параметры, но и любые другие типы данных, включая списки строк для стоп-слов, целочисленные значения для количества итераций или максимальной глубины дерева и числа типа `double` для коэффициентов сглаживания и т. п.

Пример 9.36. Образец параметров преобразователя

```
class ConfigurableWordCount(override val uid: String) extends Transformer {
  final val inputCol= new Param[String](this, "inputCol", "The input column")
  final val outputCol = new Param[String](this, "outputCol", "The output column")

  def setInputCol(value: String): this.type = set(inputCol, value)
```

```
  def setOutputCol(value: String): this.type = set(outputCol, value)

  def this() = this(Identifiable.randomUID("configurablewordcount"))

  def copy(extra: ParamMap): HardCodedWordCountStage = {
    defaultCopy(extra)
  }

  override def transformSchema(schema: StructType): StructType = {
    // Проверяем, что тип входного значения – строка
    val idx = schema.fieldIndex($(inputCol))
    val field = schema.fields(idx)
    if (field.dataType != StringType) {
      throw new Exception(
        s"Input type ${field.dataType} did not match input type StringType")
    }
    // Добавляем возвращаемое поле
    schema.add(StructField($(outputCol), IntegerType, false))
  }

  def transform(df: Dataset[_]): DataFrame = {
    val wordcount = udf { in: String => in.split(" ").size }
    df.select(col("*"), wordcount(df.col($(inputCol))).as($(outputCol)))
  }
}
```

Пользовательские оцениватели

Главное различие между интерфейсами `Estimator` и `Transformer` состоит в том, что вместо выражения явным образом преобразований над входными данными сначала производится шаг обучения в виде функции `train`. Индексатор строковых значений — один из простейших оценивателей, какие только можно реализовать, и, хотя он уже есть во фреймворке Spark, это отличная иллюстрация использования интерфейса оценивателя (пример 9.37).

Пример 9.37. Простой этап конвейера: индексатор строковых значений

```
trait SimpleIndexerParams extends Params {
  final val inputCol= new Param[String](this, "inputCol", "The input column")
  final val outputCol = new Param[String](this, "outputCol", "The output column")
}

class SimpleIndexer(override val uid: String)
    extends Estimator[SimpleIndexerModel] with SimpleIndexerParams {

  def setInputCol(value: String) = set(inputCol, value)

  def setOutputCol(value: String) = set(outputCol, value)

  def this() = this(Identifiable.randomUID("simpleindexer"))

  override def copy(extra: ParamMap): SimpleIndexer = {
```

```
    defaultCopy(extra)
  }

  override def transformSchema(schema: StructType): StructType = {
    // Проверяем, что тип входного значения – строка
    val idx = schema.fieldIndex($(inputCol))
    val field = schema.fields(idx)
    if (field.dataType != StringType) {
      throw new Exception(
        s"Input type ${field.dataType} did not match input type StringType")
    }
    // Добавляем возвращаемое поле
    schema.add(StructField($(outputCol), IntegerType, false))
  }

  override def fit(dataset: Dataset[_]): SimpleIndexerModel = {
    import dataset.sparkSession.implicits._
    val words = dataset.select(dataset($(inputCol)).as[String]).distinct
      .collect()
    new SimpleIndexerModel(uid, words)
  }
}

class SimpleIndexerModel(override val uid: String, words: Array[String])
    extends Model[SimpleIndexerModel] with SimpleIndexerParams {

  override def copy(extra: ParamMap): SimpleIndexerModel = {
    defaultCopy(extra)
  }

  private val labelToIndex: Map[String, Double] = words.zipWithIndex.
    map{case (x, y) => (x, y.toDouble)}.toMap

  override def transformSchema(schema: StructType): StructType = {
    // Проверяем, что тип входного значения – строка
    val idx = schema.fieldIndex($(inputCol))
    val field = schema.fields(idx)
    if (field.dataType != StringType) {
      throw new Exception(
        s"Input type ${field.dataType} did not match input type StringType")
    }
    // Добавляем возвращаемое поле
    schema.add(StructField($(outputCol), IntegerType, false))
  }

  override def transform(dataset: Dataset[_]): DataFrame = {
    val indexer = udf { label: String => labelToIndex(label) }
    dataset.select(col("*"),
      indexer(dataset($(inputCol)).cast(StringType)).as($(outputCol)))
  }
}
// end::SimpleIndexer[]
```

При реализации итеративного алгоритма имеет смысл задуматься об автоматическом кэшировании входных данных, которые еще не закэшированы, или о предоставлении пользователю возможности задать уровень сохранения.

Для многих алгоритмов удобнее работать со вспомогательными классами `org.apache.spark.ml.Predictor` и `org.apache.spark.ml.classification.Classifier`, а не напрямую с интерфейсом `Estimator`.

Интерфейс `Predictor` добавляет два чаще всего используемых параметра (входной и выходной столбцы в виде столбцов меток, признаков и предсказаний) и автоматически преобразует схему.

Интерфейс `Classifier` похож на интерфейс `Predictor`. Последний дополнительно включает в результат столбец `rawPredictionColumn` и предоставляет средства определения количества классов (`getNumClasses`), что удобно для задач классификации. Мы воспользуемся интерфейсом классификатора для реализации упрощенного наивного классификатора Байеса, показанного в примере 9.38. Данный интерфейс также предоставляет средства преобразования входного объекта `DataFrame` в набор RDD объектов типа `LabeledPoint`, на случай, если нужно работать с устаревшим алгоритмом в стиле библиотеки MLlib.

Пример 9.38. Простой наивный классификатор Байеса

```
// Простой наивный байесовский классификатор Бернулли.
// Для краткости контроль корректности отсутствует.
// Это лишь образец, не предназначенный для промышленной эксплуатации.
class SimpleNaiveBayes(val uid: String)
    extends Classifier[Vector, SimpleNaiveBayes, SimpleNaiveBayesModel] {

  def this() = this(Identifiable.randomUID("simple-naive-bayes"))

  override def train(ds: Dataset[_]): SimpleNaiveBayesModel = {
    import ds.sparkSession.implicits._
    ds.cache()
    // Примечание: для получения взамен [Dataset] набора RDD можно
    // воспользоваться методами getNumClasses и extractLabeledPoints.
    // Подход с набором RDD часто применяется при интеграции
    // с устаревшим кодом для машинного обучения или итеративными
    // алгоритмами, которые могут приводить к большим планам запросов.
    // Подсчитываем количество документов.
    val numDocs = ds.count
    // Получаем количество классов.
    // Заметим, что этот оцениватель предполагает, что их нумерация
    // начинается с 0 и доходит до numClasses.
    val numClasses = getNumClasses(ds)
    // Получаем количество признаков из первого столбца.
    val numFeatures: Integer = ds.select(col($(featuresCol))).head
      .get(0).asInstanceOf[Vector].size
```

```
    // Определяем количество записей для каждого класса
    val groupedByLabel = ds.select(col($(labelCol)).as[Double]).groupByKey(x => x)
    val classCounts = groupedByLabel.agg(count("*").as[Long])
      .sort(col("value")).collect().toMap
    // Выбираем метки и признаки для упрощения их отображения.
    // Примечание: мы используем для этого DataFrame с помощью
    // нетипизированного API, поскольку пользовательский тип данных
    // Vector более не является общедоступным.
    val df = ds.select(col($(labelCol)).cast(DoubleType), col($(featuresCol)))
    // Определяем ненулевую частоту каждого признака для каждой метки
    // и выводим пары «метка/индекс», используя case-класс для упрощения
    val labelCounts: Dataset[LabeledToken] = df.flatMap {
      case Row(label: Double, features: Vector) =>
        features.toArray.zip(Stream from 1)
          .filter{vIdx => vIdx._2 == 1.0}
          .map{case (v, idx) => LabeledToken(label, idx)}
    }
    // Используем типизированный API агрегирования Dataset для подсчета
    // количества ненулевых признаков для каждого индекса метки/признака
    val aggregatedCounts: Array[((Double, Integer), Long)] = labelCounts
      .groupByKey(x => (x.label, x.index))
      .agg(count("*").as[Long]).collect()
    val theta = Array.fill(numClasses)(new Array[Double](numFeatures))

    // Вычисляем знаменатель для общих априорных вероятностей
    val piLogDenom = math.log(numDocs + numClasses)
    // Вычисляем априорные вероятности принадлежности к нему
    // для каждого из классов
    val pi = classCounts.map{case(_, cc) =>
      math.log(cc.toDouble) - piLogDenom }.toArray

    // Для каждой метки/признака обновляем вероятности
    aggregatedCounts.foreach{case ((label, featureIndex), count) =>
      // Логарифм числа документов для данной метки + 2.0 (сглаживание)
      val thetaLogDenom = math.log(
        classCounts.get(label).map(_.toDouble).getOrElse(0.0) + 2.0)
      theta(label.toInt)(featureIndex) = math.log(count + 1.0) - thetaLogDenom
    }
    // Поскольку мы завершили вычисления, то удаляем из хранилища
    ds.unpersist()
    // Формируем модель
    new SimpleNaiveBayesModel(uid, numClasses, numFeatures, Vectors.dense(pi),
      new DenseMatrix(numClasses, theta(0).length, theta.flatten, true))
  }

  override def copy(extra: ParamMap) = {
    defaultCopy(extra)
  }
}

// Упрощенная модель наивного классификатора Байеса
case class SimpleNaiveBayesModel(
  override val uid: String,
  override val numClasses: Int,
  override val numFeatures: Int,
```

```
  val pi: Vector,
  val theta: DenseMatrix) extends
    ClassificationModel[Vector, SimpleNaiveBayesModel] {

  override def copy(extra: ParamMap) = {
    defaultCopy(extra)
  }

  // Приходится прибегнуть тут к некоторым ухищрениям, поскольку
  // мы выполняем вычисления с типами Vector/DenseMatrix, но вы не обязаны
  // ограничиваться нативными типами данных Spark в ваших моделях
  val negThetaArray = theta.values.map(v => math.log(1.0 - math.exp(v)))
  val negTheta = new DenseMatrix(numClasses, numFeatures, negThetaArray, true)
  val thetaMinusNegThetaArray = theta.values.zip(negThetaArray)
    .map{case (v, nv) => v - nv}
  val thetaMinusNegTheta = new DenseMatrix(
    numClasses, numFeatures, thetaMinusNegThetaArray, true)
  val onesVec = Vectors.dense(Array.fill(theta.numCols)(1.0))
  val negThetaSum: Array[Double] = negTheta.multiply(onesVec).toArray

  // Эта функциональность предсказаний, которую нужно реализовать
  // автоматически, оборачивается в преобразование ClassificationModel.
  // Если чем-то полезно транслирование модели или другие
  // оптимизации, то можете переопределить преобразование и вставить
  // сюда нужную логику.
  def predictRaw(features: Vector): Vector = {
    // Это лишь «игрушечная» реализация – вместо суммирования
    // трех векторов лучше использовать BLAS или что-то подобное,
    // но функциональность BLAS не является общедоступной
    Vectors.dense(thetaMinusNegTheta.multiply(features).toArray.zip(pi.toArray)
      .map{case (x, y) => x + y}.zip(negThetaSum).map{case (x, y) => x + y}
      )
  }
}
```

Реализация интерфейса для регрессии или кластеризации потребует использования обобщенного интерфейса `Estimator`, поскольку нет общедоступного базового набора интерфейсов, который можно было бы применить.

Если просто необходимо внести изменения в существующий алгоритм, то для доступа к нужным внутренним компонентам можно вставить свой код в пакет org.apache.spark. При этом нет никаких обычных гарантий стабильности API (от версии к версии Spark), так что при обновлении Spark следует соблюдать осторожность. Эту методику мы рассмотрим в разделе «Организация потоковой обработки с помощью Spark» на с. 282.

Дополнительные примеры лучше всего искать в самой библиотеке Spark ML (https://github.com/apache/spark/tree/master/mllib/src/main/scala/org/apache/spark/ml), и хотя большинство из них используют какие-либо внутренние API, на них можно ориентироваться при добавлении собственных функций.

Теперь, когда вы уже готовы начать расширять конвейеры Spark ML, имеет смысл создать пакет Spark, чтобы поделиться своими новыми утилитами с сообществом разработчиков (см. раздел «Использование пакетов и библиотек, созданных сообществом разработчиков» на с. 297).

Сохранение и выдача моделей и конвейеров с помощью Spark ML

Встроенные возможности сохранения моделей и конвейеров библиотеки Spark ML ограничиваются внутренним форматом Spark. Использование сохраненного конвейера требует полного SparkContext и значительных накладных расходов, так как поэлементные API предсказания пока еще не общедоступны. Функциональность экспорта PMML еще не перенесена из Spark MLlib в Spark ML, однако эти свойства поддерживаются другими внешними проектами.

Сохранение в формате PMML особенно удобно для выдачи в тех случаях, когда не хотелось бы включать все зависимости Spark, ведь оно поддерживается множеством систем. Существует внешний проект MLeap (https://github.com/TrueCar/mleap), поддерживающий экспорт множества моделей в формат PMML, да и в других проектах рассматривается возможность добавления поддержки PFA. Мы обсудим применение подобных внешних проектов в разделе «Использование пакетов и библиотек, созданных сообществом разработчиков» на с. 297.

Разработчики библиотеки Spark ML планируют расширить ее возможности сохранения, включая добавление формата PMML в SPARK-11171 (https://issues.apache.org/jira/browse/SPARK-11171), но это пока еще дело будущих версий фреймворка Spark.

Общие соображения о выдаче

Возможности сохранения обеих библиотек, как Spark MLlib, так и Spark ML, ограниченны, что усложняет применение обученной модели в динамической среде. Пользователи, у которых уже существует инфраструктура выдачи, часто вручную создают слой сохранения для своих форматов. Это приходится делать для каждого из типов моделей и требует значительных затрат времени. Подобный подход задействуется в проекте MLeap (https://github.com/TrueCar/mleap), предоставляющем свой формат сериализации моделей для множества моделей Spark с динамическим слоем выдачи. Для тех же пользователей, у которых инфраструктуры выдачи нет, существует простой и гибкий вариант: применение экспорта в формат PMML с динамическим слоем выдачи, например JPMML и API оценивания для JPMML (https://github.com/jpmml/jpmml-evaluator).

JPMML лицензирован в соответствии с AGPL, так что может оказаться неподходящим для вашей компании из-за лицензионных ограничений.

Одни из самых простых в сохранении — линейные модели, состоящие всего лишь из нескольких коэффициентов. Одна из них показана в примере 9.39.

Пример 9.39. Созданный вручную слой сохранения для GLM

```
def exportLRToCSV(model: LogisticRegressionModel) = {
  (model.coefficients.toArray :+ model.intercept).mkString(",")
}
```

Не все модели так легко сохранять, как GLM (пример 9.39), и может оказаться, что не вся информация, которую нужно сохранить, доступна непосредственно. В подобных случаях придется прибегнуть к обману и притвориться составной частью `org.apache.spark`, как мы поступим в отдельных примерах потоковой обработки (см. далее примеры 10.13 и 10.14).

Расширение (создание производных классов) моделей требуется только при работе в JVM — Py4 использует рефлексию, так что можно (и столь же небезопасно) непосредственно обращаться к внутренним данным модели для экспорта.

Резюме

Встроенные библиотеки машинного обучения фреймворка Spark поддерживают множество различных алгоритмов, но в настоящее время в основном ориентированы на пакетные сценарии использования. Можно не только непосредственно создавать модели, применяя Spark; существуют и другие инструменты машинного обучения для фреймворка, и в некоторых из них возможности выдачи моделей шире. Для обучения моделей с помощью Spark подойдут проекты Oryx (http://oryx.io/), Mahout (http://mahout.apache.org/), Sparkling Water компании H2O (https://www.h2o.ai/sparkling-water/), Algorithmia (https://algorithmia.com/) и др. В следующей главе мы рассмотрим использование компонентов Spark и всей экосистемы компонентов фреймворка.

10 Компоненты и пакеты фреймворка Spark

Spark содержит множество компонентов, ориентированных на совместную работу в виде интегрированной системы, причем многие из них распространяются как составная часть фреймворка. В этом заключается отличие Spark от экосистемы Hadoop, где для каждой задачи существуют свои проекты или системы. Вы уже научились эффективно использовать компоненты Spark Core, Spark SQL и ML, а эта глава познакомит с компонентами фреймворка Spark, предназначенными для потоковой обработки, а также внешними и созданными сообществом разработчиков компонентами (которые часто называют пакетами). Большая интегрированная система дает Spark два преимущества: упрощает как развертывание/управление кластером, так и разработку приложений благодаря снижению числа зависимостей и систем, нуждающихся в отслеживании.

Даже в ранних версиях фреймворка Spark были утилиты, которые обычно требуют координации нескольких систем, как показано на рис. 10.1.

Поскольку объекты `Dataset` и движок Spark SQL стали основой для других компонентов Spark, то после небольшой реструктуризации мы получим более актуальную версию, показанную на рис. 10.2. Она включает два из новейших компонентов Spark — библиотеку Spark ML и движок структурированной потоковой обработки (Structured Streaming). Большая часть ваших знаний о том, как работать с базовым Spark и Spark SQL, применима и к другим компонентам — хотя для каждого есть свои нюансы.

Помимо интегрированных компонентов, важные свойства во фреймворк Spark добавляют пакеты, созданные сообществом разработчиков, иногда даже заменяя встроенные возможности: в качестве примера можно привести пакет GraphFrames. Такие пакеты предоставляют разнообразную функциональность, начиная от дополнительных форматов данных (упоминавшихся в пункте «Дополнительные форматы» на с. 80) до пакетов тестирования, обсуждавшихся в главе 8, машинного обучения и алгоритмов работы с графами. Пакеты, созданные сообществом, могут

иногда становиться частью фреймворка Spark, если применяются достаточно широко, как произошло, например, с пакетом `spark-csv`, предназначенным для загрузки CSV-данных в объекты `Dataset`. В данной главе мы вкратце расскажем о том, как создать собственный пакет для сообщества, чтобы все могли воспользоваться плодами вашего труда. Хотя, конечно, никто не заставляет вас делать это.

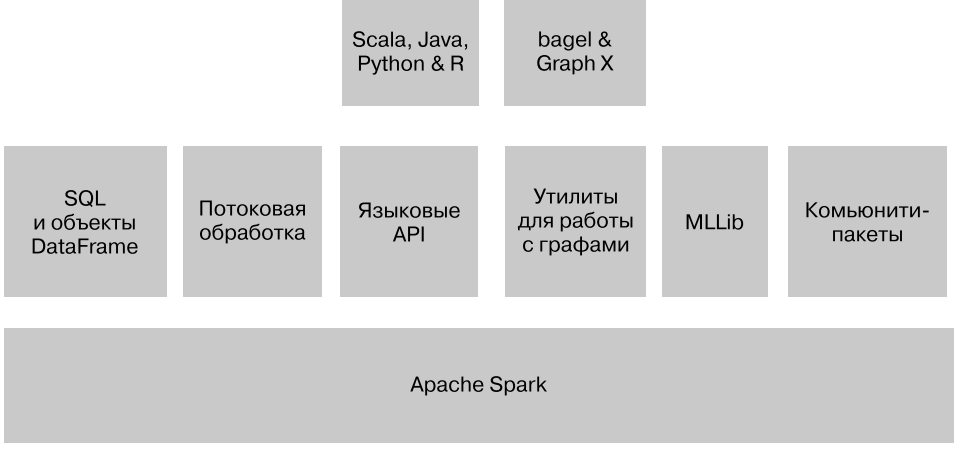

Рис. 10.1. Схема компонентов фреймворка Spark

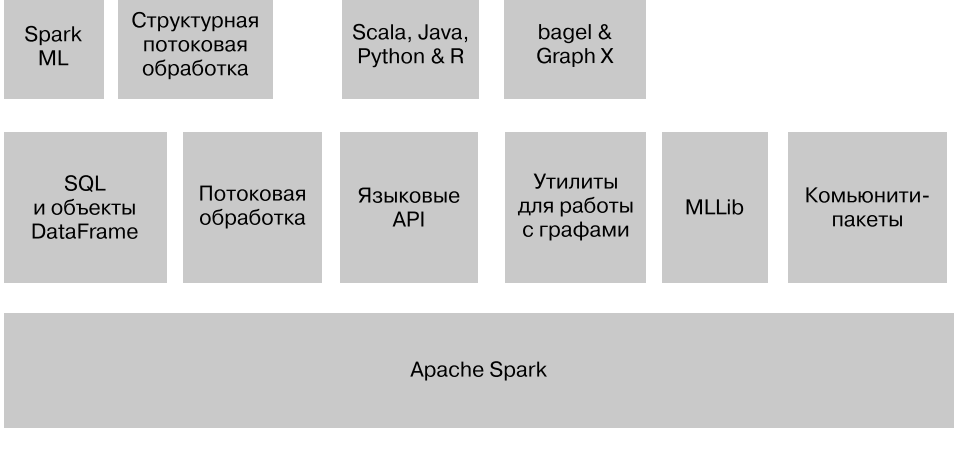

Рис. 10.2. Пересмотренная схема компонентов фреймворка Spark 2.0+

Если вы пока еще не прочитали главу 3, то рекомендуем вернуться к ней и по крайней мере бегло ее просмотреть, поскольку, как демонстрирует рис. 10.2, значительная часть новых разработок для Spark происходит на основе Spark SQL и наборов DataFrame/Dataset.

Менее заметное преимущество этого интегрированного подхода — то, что разработка любого компонента может улучшить функционирование всех остальных. Например, Spark Streaming требует значительного понижения затрат на создание задач. Данное обстоятельство повышает возможности работы Spark SQL с маленькими запросами, при которых ранее превалировали затраты на выполнение задач.

Организация потоковой обработки с помощью Spark

У Spark Streaming есть два API, один — основанный на наборах RDD, называемых `DStream`, а второй (в настоящее время находящийся на *альфа*-стадии разработки), под названием Structured Streaming, — основанный на объектах Spark SQL/`DataFrame`. Большая часть соображений относительно производительности преобразований наборов RDD и операций с объектами `DataFrame`/`Dataset` остается справедливой и в контексте потоковой обработки, и названия многих операций остаются теми же. Одни операции имеют смысл только в контексте потоковой обработки, а другие из API пакетной обработки в настоящее время не поддерживаются в API потоковой обработки. В этом разделе мы изучим некоторые специфичные для последнего API Spark нюансы, основываясь на знаниях, полученных в предыдущих главах.

Больше информации о Spark Streaming вы можете найти в книге Learning Spark Streaming, написанной Франсуа Гарильо (François Garillot) (издательство O'Reilly).

Одно из самых заметных отличий между потоковой и пакетной обработкой заключается во входных источниках данных и может привести к значительной разнице в производительности.

При разработке и тестировании потоковых приложений в локальном режиме запуск с параметрами local или local[1] в качестве ведущего узла часто вызывает проблемы. Spark Streaming требует много работников для успешного функционирования, так что при тестировании лучше использовать большее число работников, например local[4].

Источники и приемники данных

Вне зависимости от используемой версии API Spark Streaming имеются определенные нюансы того, как этот компонент потребляет и выводит данные. Хорошо секционированные входные данные могут существенно повлиять на ситуацию, как и секционирование входных наборов RDD и `DataFrame`. Неправильное секциони-

рование или неправильно настроенные источники данных могут привести к узким местам при чтении данных, равно как и нежелательным их потерям.

Оба API Spark Streaming непосредственно поддерживают файловые источники данных для тестирования, автоматически «подхватывающие» новые подкаталоги, и источники в виде «сырых» сокетов. Как для структурированной потоковой обработки, так и более ошибкоустойчивого API `DStream` фреймворка Spark поддерживается (в предварительном варианте) источник Kafka.

Для простого тестирования Spark позволяет задействовать потоки в оперативной памяти, объекты `QueueStream` для потоков `DStream`, а также `MemoryStream` для потоковых наборов `Dataset`. Объекты `QueueStream` фреймворка не поддерживают многие операции (например, `groupByWindow`), что ограничивает возможные сценарии их применения. Вместо объектов `QueueStream` многие используют для тестирования пакет spark-testing-base, как в пункте «Потоковая обработка» на с. 231.

В API DStream имеются дополнительные источники данных от Apache, но они скомпонованы в виде отдельных файлов JAR во избежание чрезмерного увеличения размера и проблем с лицензированием. Среди этих источников — Kafka, Flume, Kinesis, Twitter, ZeroMQ и MQTT. Руководства по интеграции существуют для источников Kafka (http://spark.apache.org/docs/latest/streaming-kafka-integration.html), Flume (http://spark.apache.org/docs/latest/streaming-flume-integration.html) и Kinesis (http://spark.apache.org/docs/latest/streaming-kinesis-integration.html). Часть из них будет включена в Structured Streaming в грядущих версиях, причем некоторые, возможно, в виде пакетов, созданных сообществом разработчиков.

Некоторые источники данных были исключены из Spark и теперь доступны в проекте Apache Bahir (http://bahir.apache.org/). Этот проект и другие потоковые источники данных в пакетах Spark можно включить в свой проект, следуя инструкциям из раздела «Использование пакетов и библиотек, созданных сообществом разработчиков» на с. 297.

На момент написания данной книги большинство источников данных Structured Streaming пока еще недоступны. Дополнительные источники, вероятно, будут доступны аналогичным API `DStream` образом в будущих версиях Structured Streaming и в виде пакетов фреймворка Spark, которые мы рассмотрим в разделе «Использование пакетов и библиотек, созданных сообществом разработчиков» на с. 297.

Получатели данных

Значительная часть источников `DStream` фреймворка Spark использует выделенные процессы-получатели для чтения данных из потоковых источников, с заслуживающими упоминания исключениями в виде файловых источников и «лишенных получателя» источников Kafka. В основанной на получателях схеме определенное количество работников настроены таким образом, чтобы читать данные из входных потоков данных, которые — при недостаточно тщательной конфигурации — могут

стать узкими местами. Неподходящая конфигурация получателя способна существенно ограничить объем данных, которые может обработать Spark. Нюансы каждого конкретного получателя зависят от источника данных, так что обязательно прочитайте документацию по вашему получателю.

У ряда источников данных есть несколько вариантов чтения. Например, в Kafka применяется как основанная на получателях конфигурация, так и «прямой» подход, при котором отдельные исполнители Spark непосредственно читают данные из Kafka.

Повторное секционирование

Как и в случае обычных наборов RDD и объектов `DataFrame`, входные источники данных могут быть секционированы неоптимальным для обработки образом с помощью Spark. При основанном на получателях подходе секционирование потока `DStream` обычно отражает конфигурацию получателя. В непосредственных (лишенных получателя) подходах начальное секционирование основывается на секционировании входного потока данных, подобно тому как начальное секционирование наборов RDD аналогично секционированию HDFS.

Если секционирование ваших данных не идеально подходит для обработки, то устранить проблему легче всего с помощью повторного секционирования явным образом аналогично RDD. Оно может оказаться более простым решением, чем настройка и объединение нескольких получателей данных, и напоминает работу с наборами RDD (пример 10.1). При повторном секционировании пар «ключ — значение» наличие известного объекта `Partitioner` способно значительно ускорить многие операции, как обсуждалось в главе 6.

Пример 10.1. Повторное секционирование потока DStream

```
dstream.repartition(20)
```

Однако если источником узкого места является чтение данных, то повторного секционирования после первоначального чтения может оказаться недостаточно — придется повысить или количество секций во входном источнике данных, или количество получателей.

Этот пример может показаться очень знакомым. Дело в том, что, подобно многим операциям над потоками `DStream`, он реализован с помощью функции `transform`. Она принимает на входе набор RDD для каждого промежутка времени и применяет к нему преобразования. Представление с применением промежутков времени функции `transform` позволяет при необходимости повторно задействовать для потоков `DStream` операцию или пользовательскую функцию, предназначенную для работы

с наборами RDD. В примере 10.2 мы делаем вид, что метод `repartition` отсутствует в классе `DStream`, и создаем альтернативную функцию.

Пример 10.2. Повторное секционирование потока DStream с помощью функции transform

```
def dStreamRepartition[A: ClassTag](dstream: DStream[A]): DStream[A] = {
  dstream.transform{rdd => RDD.repartition(20)}
}
```

Повторное секционирование может быть полезно даже при использовании прямого получателя данных Kafka на основе DStream. Оно пригодится в случае, когда количества секций Kafka недостаточно для достижения нужной степени параллелизма.

Интервалы времени между пакетами

Интервалы времени между пакетами отражают традиционный компромисс между пропускной способностью и длительностью задержки. Spark Streaming полностью использует каждый интервал времени, прежде чем начать отсчет следующего. А значит, нужно задавать значение интервала времени между пакетами достаточно большим для того, чтобы предыдущий пакет был обработан до запланированного запуска следующего.

Время обработки пакета зависит от конкретного приложения, так что дать какие-либо общие рекомендации затруднительно. Распространенная практика — начинать с большого интервала между пакетами и уменьшать его до тех пор, пока время обработки микропакета не приблизится к длительности интервала между пакетами, после чего вернуться к последнему безопасному значению интервала.

При изменении интервала между пакетами желательно убедиться, что длительность оконных операций, обсуждаемых в подразделе «Соображения относительно потоков DStream» на с. 286, кратна ему.

Интервал между пакетами задается по-разному для API потоков `DStream` и Structured Streaming. В API `DStream` он устанавливается на уровне приложения/контекста, как показано в примере 10.3. А настройки API Structured Streaming задаются для каждого выводимого результата каждого запроса, как будет показано в примере 10.7.

Пример 10.3. Создание объекта StreamingContext с интервалом между пакетами в 1 секунду

```
val batchInterval = Seconds(1)
new StreamingContext(sc, batchInterval)
```

Интервалы создания контрольных точек

Истоки сильнейших сторон фреймворка Spark — в информации, содержащейся в DAG (наборов RDD) или планах запросов (объектов `DataFrame`/`Dataset`), и оптимизациях, которые Spark производит благодаря ей. Как и в случае итеративных алгоритмов, потоковые операции могут генерировать графы DAG или планы запросов, слишком большие для того, чтобы поместиться в оперативной памяти драйверной программы. Для простых операций, не зависящих от предыдущих пакетов, таких как `map`, `filter` или `join`, вероятность возникновения проблемы при потоковой обработке вряд ли выше, чем при пакетной. Для операций же, зависящих от предыстории, например потоковых операций агрегирования наборов `Dataset` и `updateStateByKey` потоков `DStream`, в целях предотвращения чрезмерного роста размеров DAG или плана запроса необходимо создать контрольные точки.

В Spark данные и операции являются распределенными, но в драйверной программе должен храниться весь DAG или план запроса. Утверждение может показаться нелогичным, ведь Spark — распределенная система, но на самом деле это ключ к обеспечению отказоустойчивости фреймворка Spark с помощью повторных вычислений.

Чтобы включить механизм создания контрольных точек для потоковой обработки, необходимо вызвать функцию `checkpoint`, передав ей путь к каталогу, который Spark будет использовать для хранения контрольных точек. Подробности см. в пункте «Создание контрольных точек» на с. 142.

Создание контрольных точек для метаданных используется при восстановлении после сбоев и обсуждается в подразделе «Режим высокой доступности (отказы драйвера и создание контрольных точек)» на с. 296.

Соображения относительно потоков DStream

В основе API потоков DStream Spark Streaming лежит API наборов RDD, а большинство его операций просто оборачивают методы RDD в операцию `transform`. Для этих преобразований в основном справедливы те же соображения относительно производительности, которые мы обсуждали в главе 5. Однако не все функции представляют собой простые адаптеры для операций над наборами RDD, самая очевидная из функций, не имеющих прямых аналогов в API наборов RDD, — операция `window`, позволяющая формировать скользящее временно*е* окно для входного потока `DStream`, и ее аналоги (например, `reduceByWindow`).

Как и в API наборов RDD Spark Scala, многие функции были добавлены с помощью неявных преобразований в API Scala. Подобно положению дел в API наборов RDD Spark Java, специальные функции возвращают различные типы для поддержки конкретных операций (например, объекты JavaPairDStream для данных типа «ключ — значение»).

Оконные операции позволяют проводить вычисления по последним K пакетам данных, что очень удобно для, например, вычисления скользящего среднего или калмановских фильтров. По сути, оконные операции определяются с помощью `windowDuration` (ширина окна) и `slideDuration` (частота вычисления окна).

Один из важнейших для оконных операций аспектов — предоставление Spark возможности инкрементно вычислять окна с помощью функции `reduceByKeyAndWindow` с функциями `reduceFunc` и `invReduceFunc` в качестве параметров. Конечно, это допустимо только при наличии легко выразимой обратной операции для вашей свертки (как операция `-` для `+`).

Операции вывода. Встроенные свойства сохранения объектов `DStream` несколько ограниченны: имеется встроенная поддержка только для сохранения отдельных пакетов в виде объектных или текстовых файлов. Пример 10.4 иллюстрирует сохранение текстового файла.

Использовать объектные файлы не рекомендуется, поскольку они зависимы от Java-сериализации объектов и отличаются от последовательных файлов. Может оказаться, что загрузка объектных файлов, работавшая нормально в одной версии Spark, перестанет работать в другой.

Пример 10.4. Простое сохранение в виде текстовых файлов

```
dstream.saveAsTextFiles(target)
```

Можно сохранять и в другие форматы, используя операцию `foreachRDD` для попакетного вывода с помощью API наборов RDD. Операция `foreachRDD` функционирует почти так же, как и `transform`, но является скорее действием, а не преобразованием. Существует несколько возможных вариантов API `foreachRDD` в зависимости от того, требуется ли для ваших действий информация о самом пакете (допустим, время), а не только его содержимое (скажем, для записи в различные каталоги), как показано в примере 10.5.

Пример 10.5. Сохранение в виде последовательных файлов с помощью операции foreachRDD

```
dstream.foreachRDD{(rdd, window) =>
  rdd.saveAsSequenceFile(target + window)
}
```

Соображения относительно Structured Streaming

В Structured Streaming появилась новая модель потоковой обработки данных в Spark, более тесно связанная с лежащими в ее основе табличными абстракциями пакета Spark SQL. Structured Streaming можно представлять как запуск SQL-запросов на бесконечной таблице с добавляемыми потоком записями[1]. В отличие от основанного на наборах RDD API, Structured Streaming не вводит новый тип данных (такой как `DStream`), а продолжает использовать существующий тип `Dataset`, добавляя в него булево значение `isStreaming`, чтобы можно было различать потоковый и пакетный `Dataset`.

Structured Streaming появился лишь в Spark 2.0 и на момент написания данной книги (Spark 2.1) еще не может считаться готовым к промышленной эксплуатации.

В отличие от потоков DStream, структурированные потоки можно создавать с помощью обычного объекта SparkSession.

Потоковые объекты `Dataset` поддерживают множество разнообразных операций, но не все операции, реализованные в пакетных `Dataset`, поддерживаются в потоковых. Не поддерживаются обычно те операции, которые не имеют смысла — например, в Spark 2.0 функция `toJson()` (служащая для преобразования объекта `Dataset` в объект `Dataset` в формате JSON) не поддерживается, поскольку реализация преобразует `Dataset` в набор RDD. Конечно, трудно понять, реализована ли внутренне та или иная операция с помощью преобразования RDD.

Пакет Structured Streaming реализован с помощью непрерывных объектов Dataset. Однако не все поддерживаемые операции над объектами Dataset доступны для непрерывных Dataset, и возможность проверки во время компиляции для потоковой обработки отсутствует.

Источники данных

Structured Streaming в настоящее время поддерживает очень ограниченный набор источников данных, хотя в будущих версиях ожидается пополнение этого списка — как в самом фреймворке Spark, так и в виде пакетов. Загрузка потокового

[1] Spark SQL можно использовать с API DStream, но это весьма непростая задача.

источника данных не особо отличается от загрузки обычных SQL-данных: нужно просто вызвать функцию `readStream` вместо `read` (пример 10.6).

Пример 10.6. Простой завершенный образец использования режима ввода

```
session.readStream.parquet(inputPath)
```

Логический вывод схемы выборки с потоковыми данными не работает, так что при необходимости загрузить нечто подобное JSON, придется задавать схему вручную, как при работе с объектами Dataset.

Хотя текущий набор источников данных для Structured Streaming оставляет желать лучшего, работы по добавлению новых форматов продвигаются достаточно быстро (см. SPARK-15406 (https://issues.apache.org/jira/browse/SPARK-15406))[1]. В качестве временного решения (только для целей разработки) текущий API `DStream` способен записывать данные в хранилище `HDFS` в формате Parquet путем операции `foreachRDD`, а каталог `parquet` можно задействовать как входной для Structured Streaming.

Операции вывода

Для записи в приемники данных в Structured Streaming используется класс `DataStreamWriter` (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.streaming.DataStreamWriter) — не такой, как в обычных `DataFrame`/`Dataset`. Хотя записывающий класс и отличается, общие принципы не поменялись. Важная и обязательная настройка — особый параметр `outputMode`, который может принимать значения `append` (для новых строк) и `complete` (для всех строк). В настоящее время для всех потоковых `Dataset` с операциями агрегирования должен использоваться режим `complete`, так что нет способа непосредственно получить только изменившиеся сводные показатели (как в примере 10.7).

Пример 10.7. Простой завершенный образец использования режима вывода

```
val query = counts.writeStream.
  // Задаем значение режима вывода Complete для поддержки агрегирования
  outputMode(OutputMode.Complete())
  // Выводим результат в формате parquet
  format("parquet").
  // Задаем интервал, с которым будут «подхватываться» новые данные
  trigger(ProcessingTime(1.second))
  queryName("pandas").start()
```

Класс `DataStreamWriter` может выводить коллекции и в другие форматы, помимо показанного в примере `Parquet`. В число встроенных форматов входят `console`,

[1] И действительно, в промежутке между двумя редакциями данной книги вышла младшая альфа-версия Spark, в которой в Structured Streaming была добавлена поддержка Kafka.

`foreach` и `memory`. Первый из них выводит результаты в терминал, а последний записывает их в локальную таблицу. Формат `foreach` отличается тем, что его нельзя определить с помощью функции `format`, а следует вызывать метод `foreach` объекта `writer` для установки нужной функции.

В Spark 2.0 консольный режим вывода приводит к возврату всего потока локально.

Пользовательские приемники данных

Если говорить не только о стандартных приемниках данных, то может понадобиться выполнить какие-либо произвольные вычисления над результатом потока, помимо вывода в один из приемников данных по умолчанию. Так как необходимо указывать имя пользовательского приемника данных, затруднительно включать в него произвольные функции: функцию для заданного приемника данных следует знать во время компиляции (то есть она не может меняться в зависимости от вводимых пользователем данных). В примере 10.8 для этого применяется функция, которую можно задать с помощью текущего общедоступного API записи аналогично тому, как и для других приемников данных (как показано в примере 10.9). Этим можно воспользоваться для более широкой эмуляции поведения операции `foreachRDD`, обратившись к внутренним компонентам, которые мы рассмотрим в пункте «Машинное обучение с помощью Structured Streaming» на с. 291.

Текущие доступные в Structured Streaming API приемников данных предполагают микропакетную обработку, но они, вероятно, будут меняться, поскольку одна из целей Structured Streaming состоит в том, чтобы позволить отказаться от микропакетной обработки в механизме выполнения.

Пример 10.8. Пользовательский приемник данных для Structured Streaming

```
/**
 * Простой пользовательский приемник данных, предназначенный для
 * иллюстрации того, как в настоящее время предполагается
 * задействовать API пользовательских приемников данных
 */
class BasicSinkProvider extends StreamSinkProvider {
  // Наш приемник данных очень прост, поэтому мы тут ничего не делаем,
  // но здесь может быть код для настройки
  override def createSink(
      sqlContext: SQLContext,
      parameters: Map[String, String],
      partitionColumns: Seq[String],
```

```
      outputMode: OutputMode): BasicSink = {
    new BasicSink()
  }
}

class BasicSink extends Sink {
  /*
   * В соответствии со SPARK-16020 произвольные преобразования
   * не поддерживаются, но преобразование в набор RDD творит чудеса
   */
  override def addBatch(batchId: Long, data: DataFrame) = {
    val batchDistinctCount = data.rdd.distinct.count()
    println(s"Batch ${batchId}'s distinct count is ${batchDistinctCount}")
  }
}
```

Пример 10.9. Вывод данных в простой пользовательский приемник данных для Structured Streaming

```
ds.writeStream.format(
  "com.highperformancespark.examples.structuredstreaming." +
    "BasicSinkProvider")
  .queryName("customSinkDemo")
  .start()
```

В Spark 2.0 внутри приемника данных не допускается выполнение никаких операций, способных привести к изменению логического плана (например, SQL/DataFrame/Dataset преобразований). В качестве временного решения можно преобразовать объект DataFrame в набор RDD и выполнить нужное преобразование над RDD.

Машинное обучение с помощью Structured Streaming

В первую версию пакета Structured Streaming API для машинного обучения еще не были интегрированы. Однако при некоторой изобретательности можно было заставить работать на основе Structured Streaming собственные алгоритмы машинного обучения. Предварительный экспериментальный вариант интеграции Structured Streaming и машинного обучения доступен в репозитории spark-structured-streaming-ml на GitHub (https://github.com/holdenk/spark-structured-streaming-ml), но следует отметить, что он предназначен не для промышленной эксплуатации, а лишь для иллюстративных целей.

Следить за прогрессом в поддержке Structured Streaming конвейерами библиотеки Spark ML можно в SPARK-16424 (https://issues.apache.org/jira/browse/SPARK-16424).

Один из простейших потоковых алгоритмов машинного обучения, которые только можно реализовать на основе Structured Streaming, — наивный классификатор Байеса, поскольку значительная доля необходимых для него вычислений сводится к группировке и агрегированию. После написания алгоритма для обучения модели возникает интересный вопрос: как выполнить сбор агрегированных данных таким образом, чтобы их можно было использовать для предсказания. В пакете Structured Streaming фреймворка Spark есть табличный (в оперативной памяти) формат вывода, пригодный для хранения агрегированных результатов, как показано в примере 10.10.

Пример 10.10. Сводные показатели для наивного классификатора Байеса (Structured Streaming)

```
// Подсчитываем количество с помощью преобразования объекта Dataset
val counts = ds.flatMap{
  case LabeledPoint(label, vec) =>
    vec.toArray.zip(Stream from 1).map(value => LabeledToken(label, value))
}.groupBy($"label", $"value").agg(count($"value").alias("count"))
  .as[LabeledTokenCounts]
// Создаем таблицу для хранения результатов
val randomId = java.util.UUID.randomUUID.toString.filter(_ != '-').toString
val tblName = "qbsnb" + randomID
// Выводим агрегированные результаты в режиме complete в таблицу в памяти
val query = counts.writeStream.outputMode(OutputMode.Complete())
  .format("memory").queryName(tblName).start()
val tbl = ds.sparkSession.table(tblName).as[LabeledTokenCounts]
val model = new QueryBasedStreamingNaiveBayesModel(tbl)
```

Пока что приведенный пример еще не включает все данные, необходимые для создания экземпляра обычной наивной модели Байеса фреймворка Spark. Можно или заняться добавлением дополнительных сводных показателей (для числовых записей и т. д.), или попробовать другой подход.

Изначальный наш подход к наивному классификатору Байеса плохо поддается обобщению на те алгоритмы, которые сложно представить в виде операций агрегирования на объектах `Dataset`. В данном случае может оказаться полезным взглянуть на то, как машинное обучение было реализовано в предварительном потоковом API Spark, основанном на потоках `DStream`. Реализация метода `foreachRDD` объектов `DStream` позволяет получить доступ к нижележащему микропакетному представлению данных, конечно, при наличии механизма обновления для включения новых данных в существующую модель. К сожалению, прямого аналога метода `foreachRDD` в Structured Streaming не существует, но добиться схожего поведения можно с помощью пользовательского приемника данных (как показано в примере 10.11).

Пример 10.11. Простейший пользовательский приемник данных для обучения наивного классификатора Байеса

```
object SimpleStreamingNaiveBayes {
  val model = new StreamingNaiveBayes()
}

class StreamingNaiveBayesSinkprovider extends ForeachDatasetSinkProvider {
```

```
  override def func(df: DataFrame) {
    val spark = df.sparkSession
    SimpleStreamingNaiveBayes.model.update(df)
  }
}
```

Как и при записи объектов `DataFrame` в пользовательские форматы, для применения сторонних приемников данных можно указывать в качестве параметра функции `format` полное имя класса. Пока библиотека Spark ML еще ожидает интеграции с Structured Streaming, пользовательский приемник данных послужит для интеграции машинного обучения в Structured Streaming, как показано в примере 10.12.

Пример 10.12. Вывод в пользовательский приемник данных для обучения наивного классификатора Байеса

```
// Обучаем с помощью модели из объекта SimpleStreamingNaiveBayes.
// При вызове для нескольких потоков все потоки обновляют параметры
// одной модели :(
// или, точнее, обновляли бы, если бы мы не «зашили» жестко в код
// имя запроса, предотвращая запуск нескольких одинаковых потоков.
def train(ds: Dataset[_]) = {
  ds.writeStream.format(
    "com.highperformancespark.examples.structuredstreaming." +
      "StreamingNaiveBayesSinkProvider")
    .queryName("trainingnaiveBayes")
    .start()
}
```

Этот простой пользовательский приемник данных следует полностью определить на момент компиляции, поскольку он формируется с помощью конкретного названия. Если проблемы при обновлении Spark вас не пугают, то можете углубиться во «внутренности» фреймворка и создать приемник данных, который бы вел себя более похоже на метод `foreachRDD` (как показано в примере 10.13).

Пример 10.13. «Нехороший» пользовательский приемник данных для Structured Streaming

```
/**
 * Создаем пользовательский приемник данных аналогично старому доброму foreachRDD.
 * Передаваемая функция вызывается для каждого промежутка времени
 * с объектом Dataset, соответствующим этому промежутку, в качестве параметра.
 *
 * Эта функция должна потреблять объект Dataset (например, с помощью
 * вызова метода "foreach" или "collect").
 * Согласно SPARK-16020, произвольные преобразования не поддерживаются,
 * но преобразование типа в набор RDD позволяет использовать и другие
 * преобразования, кроме "foreach" или "collect", сохраняя инкрементность.
 *
 */
abstract class ForeachDatasetSinkProvider extends StreamSinkProvider {
  def func(df: DataFrame): Unit

  def createSink(
      sqlContext: SQLContext,
```

```
      parameters: Map[String, String],
      partitionColumns: Seq[String],
      outputMode: OutputMode): ForeachDatasetSink = {
    new ForeachDatasetSink(func)
  }
}

/**
 * Пользовательский приемник данных, подобный старому доброму foreachRDD.
 * В момент применения с записывающим классом не следует формировать его
 * непосредственно, вместо этого лучше создать подкласс класса
 * [[ForeachDatasetSinkProvider]] и передать формат объекту
 * класса DataStreamWriter фреймворка Spark.
 * Можно использовать и непосредственно, как в StreamingNaiveBayes.scala.
 */
case class ForeachDatasetSink(func: DataFrame => Unit)
    extends Sink {

  override def addBatch(batchId: Long, data: DataFrame): Unit = {
    func(data)
  }
}
```

Поскольку для формирования приемника данных недостаточно имени класса, то необходимо передать сам приемник данных в Structured Streaming. Именно это, вероятнее всего, перестанет работать в следующих версиях фреймворка Spark[1].

Примеры 10.14 и 10.15 демонстрируют небезопасный способ выполнения запросов.

Пример 10.14. «Нехороший» потоковый диспетчер запросов, позволяющий запускать собственные запросы (небезопасным образом)

```
package org.apache.spark.sql.streaming

import scala.collection.mutable

import org.apache.spark.sql._
import org.apache.spark.sql.execution.streaming.Sink
/**
 * :: Экспериментальный ::
 * Класс для управления всеми действующими в [[SparkSession]] запросами
 * [[StreamingQuery]].
 *
 * @с версии 2.0.0
 */
case class EvilStreamingQueryManager(streamingQueryManager: StreamingQueryManager)
{
  def startQuery(
    userSpecifiedName: Option[String],
```

[1] Хотя существует проект запроса (https://github.com/apache/spark/pull/14691) на включение изменений, нацеленный на то, чтобы нечто подобное стало возможным и с интерфейсным API.

```
    userSpecifiedCheckpointLocation: Option[String],
    df: DataFrame,
    sink: Sink,
    outputMode: OutputMode): StreamingQuery = {
    streamingQueryManager.startQuery(
      userSpecifiedName,
      userSpecifiedCheckpointLocation,
      df,
      sink,
      outputMode)
  }
}
```

Пример 10.15. Применение ESQM для непосредственного запуска запроса с помощью пользовательского приемника данных

```
def evilTrain(df: DataFrame): StreamingQuery = {
  val sink = new ForeachDatasetSink({df: DataFrame => update(df)})
  val sparkSession = df.sparkSession
  val evilStreamingQueryManager = EvilStreamingQueryManager(sparkSession.streams)
  evilStreamingQueryManager.startQuery(
    Some(s"snb-train-$uid"),
    None,
    df,
    sink,
    OutputMode.Append())
}
```

Если вас интересует непосредственная поддержка аналога foreachRDD в Structured Streaming или поддержка других пользовательских приемников данных, то отслеживайте прогресс SPARK-16407 (https://issues.apache.org/jira/browse/SPARK-16407).

Хотя API Structured Streaming пока что явно не готов для промышленной эксплуатации, он предлагает немало возможностей своего расширения для поддержки машинного обучения.

Состояние потока и отладка

Одна из специфических для потоковой обработки задач — выяснение статуса источников и приемников данных. Каждый запрос связан по крайней мере с одним приемником данных, но, возможно, с несколькими источниками. Функция `status` объекта `StreamingQuery` возвращает объект с информацией о статусе запроса и всех связанных с ним источников данных. А функция `toString()` позволяет получить этот объект в красиво отформатированном и удобочитаемом виде.

При работе с наборами RDD обычно выводят небольшой поднабор данных, а для потоковых `DataFrame` для достижения того же результата используются консольные

приемники данных. Если задействовать объект `ConsoleSink` для входных маркированных точек данных, показанных в примере 10.11, то в результате получим пример 10.16.

Пример 10.16. Блокирующий консольный приемник данных

```
labeledPoints.writeStream.format("console").start().processAllAvailable()
```

Из-за ограничений Spark 2.0 при использовании консольного приемника данных производится отправка (с помощью collect()) всего пакета на драйвер, так что его нельзя применять для больших наборов данных. Если ваши наборы велики, то лучше ограничить объем данных до записи в консольный приемник.

Режим высокой доступности (отказы драйвера и создание контрольных точек)

Большинство приложений Spark исходят из допущения о невозможности отказа драйверной программы, хотя признают вероятность отказа и восстановления любого числа работников/исполнителей. Но для выполняемых в течение очень длительного времени заданий, таких как потоковая обработка, допущение о безотказной работе драйверной программы может оказаться неправильным. Более того, в случае потоковой обработки простой перезапуск задания не выход из ситуации, поскольку способен привести к потере данных. Режим высокой доступности функционирует на основе создания контрольных точек состояния драйвера и обеспечивает восстановление Spark в случае отказа драйверной программы. Конкретный механизм перезапуска зависит от режима развертывания, однако требуемые изменения кода остаются одними и теми же в любом случае. Для успешного перезапуска нужно передать функцию, реализующую восстановление (пример 10.17), вместо создания объекта `StreamingContext`, показанного в примере 10.3.

Пример 10.17. Создание контрольных точек для метаданных/восстановление потокового контекста

```
def createStreamingContext(): StreamingContext = {
  val batchInterval = Seconds(1)
  val ssc = new StreamingContext(sc, batchInterval)
  ssc.checkpoint(checkpointDir)
  // Далее создаем все нужные потоки
  // и все требуемые для них отображения.
  ssc
}
val ssc = StreamingContext.getOrCreate(checkpointDir,
  createStreamingContext _)
// Делаем все, что нужно, независимо от состояния.
// Запускаем контекст и начинаем выполнение.
   ssc.start()
```

В настоящее время накопители и транслирующие переменные не восстанавливаются в режиме высокой доступности. Их использование приведет к видимости работы программы вплоть до того момента, когда понадобится восстановление из контрольной точки. В качестве обходного пути можно применить компонент-одиночку, но значения не будут восстановлены.

GraphX

GraphX — устаревший компонент Apache Spark, более не обновляемый. Он имеет множество существенных проблем с производительностью, а в ряде случаев выполняет итеративные вычисления, не создавая контрольную точку, что требуется для очистки DAG фреймворком Spark. На момент написания данной книги самая многообещающая альтернатива GraphX — пакет GraphFrames (http://graphframes.github.io/), созданный сообществом разработчиков, включение которого в ваши приложения мы обсудим ниже.

Использование пакетов и библиотек, созданных сообществом разработчиков

Помимо поставляемых вместе со Spark компонентов, существует огромное множество предназначенных для Spark пакетов, созданных сообществом. Некоторые из них вы уже встречали в данной книге, включая spark-testing-base (https://github.com/holdenk/spark-testing-base), spark-csv (https://github.com/databricks/spark-csv) и spark-avro (https://github.com/databricks/spark-avro). Среди других заслуживающих упоминания пакетов для Spark можно назвать GraphFrames (http://graphframes.github.io/), рассматриваемый многими как преемник GraphX, и Apache Bahir (http://bahir.apache.org/), предоставляющий набор расширений, предназначенных для работы с Apache Spark, — в основном форматов входных данных. Перечень пакетов Spark, каталогизированный по области применения, можно найти на https://spark-packages.org/ (рис. 10.3).

Помимо общедоступных пакетов, аналогичным образом можно добавить и имеющиеся в вашей компании внутренние библиотеки, которые вы хотели бы включить в свои приложения Spark или использовать в командной строке фреймворка.

Пакеты Spark не проходят какой-либо тщательной проверки перед выпуском, так что придется самим оценивать степень их пригодности перед использованием.

Механизмы, применяемые для включения пакетов с сайта spark-packages.org из командной строки и с помощью сценария spark-submit, подходят и для любых опубликованных в Maven артефактов.

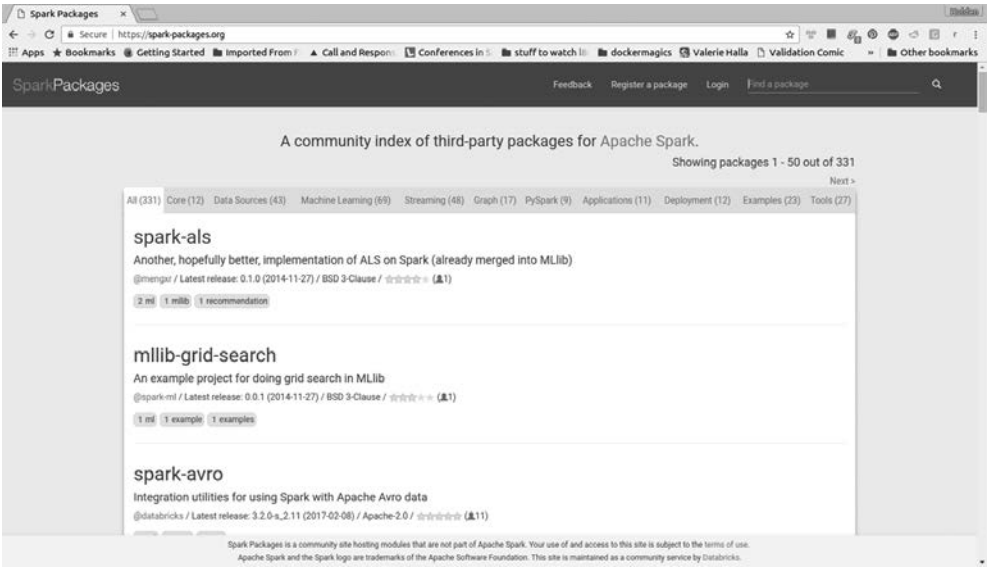

Рис. 10.3. Сайт с пакетами для фреймворка Spark

Возможно, в начале работы с новым пакетом вам захочется поэкспериментировать с ним из командной строки Spark. К счастью, механизм включения пакетов с помощью сценария `spark-submit` работает для `spark-shell` ничуть не хуже: достаточно лишь добавить параметр `--packages [maven_coordinates]` в команду запуска командной оболочки.

В первых версиях фреймворка Spark некоторые из этих способов приводили к тому, что пакет включался лишь в переменную среды classpath исполнителя, в данном случае его можно было вручную добавить в переменную среды classpath драйвера.

Добавить в проекты необходимый во время компиляции Java/Scala пакет можно, просто скопировав его Maven-координаты и включив в сборку. Если вы уже включили другие библиотеки, то удобно будет собрать все зависимости в одном обобщающем JAR-файле. Но при необходимости изменить версию пакетов во время выполнения с помощью сценария `spark-submit` просто пометьте их как имеющиеся.

Экосистема пакетов Spark не ограничивается JVM. Однако среди не-JVM-языков программирования (таких как Python и R) библиотеки чаще распространяются с помощью утилит, ориентированных на конкретный язык (например, `PyPi` и `CRAN`). Больше информации о включении пользовательских библиотек на других языках программирования можно найти в главе 7.

С пакетами Spark в настоящее время поставляются файлы .pyc только для Python 2.7, но не для линейки Python 3.X; упомянутый факт делает эту версию не совсем идеальной в глазах пользователей Python. Надеемся, в будущем данная ситуация изменится. А пока что это означает сложности в применении GraphFrames с Python 3.

Создание пакета Spark. Пакеты Spark позволяют обычным людям, не имеющим отношения к проекту Apache Spark, выпускать свои основанные на Spark усовершенствования и библиотеки. Некоторые пакеты Spark также разработаны участниками проекта Apache Spark, например spark-testing-base или Bahir. Создавая пакет Spark, вы даете другим разработчикам возможность использовать свой код с помощью `spark-submit` или командной оболочки Spark и находить его на сайте Spark Packages (https://spark-packages.org/). Все преимущества, связанные с созданием пакета Spark, за исключением перечисления версий, можно получить и за счет публикации в централизованной системе Maven — а это намного проще.

Первый шаг — зарегистрироваться на сайте Spark Packages (https://spark-packages.org/), воспользовавшись учетной записью GitHub, в которой располагается ваш пакет. Если вы хотели бы увидеть на этом сайте перечисление последних выпущенных версий, то нужно опубликовать пакет как в централизованной системе Maven, так и на сайте Spark Packages. Простейший способ публикации и сборки пакетов Spark — использовать плагин sbt-spark-package (https://github.com/databricks/sbt-spark-package). Он способен упростить управление зависимостями компонентов Spark, как показано в подразделе «Управление зависимостями Spark» на с. 50. Для добавления этого плагина в сборку SBT необходимо создать файл `project/plugins.sbt` и вставить в него код из примера 10.18.

Пример 10.18. Включение плагина sbt-spark-package в project/plugins.sbt

```
resolvers += ["Spark Package Main Repo" at
  "https://dl.bintray.com/spark-packages/maven"]

addSbtPlugin("org.spark-packages" % "sbt-spark-package" % "0.2.5")
```

После добавления этого плагина в сборку для публикации нового пакета остается настроить всего несколько аспектов. Необходимо задать значение `spName` равным имени проекта GitHub, `sparkVersion` — равным целевой версии Spark, а `sparkComponents` — списку используемых компонентов. Образец конфигурации, основанный на упрощенной версии файла сборки из проекта spark-testing-base, показан в примере 10.19.

Пример 10.19. Образец конфигурации пакета sbt-spark-package

```
sparkVersion := "2.1.0"
sparkComponents ++= Seq("core")
spName := "holdenk/spark-testing-base"
```

Если ваш пакет зависит от внутреннего устройства Spark, то может понадобиться выполнить его кросс-сборку для нескольких версий Spark. Встроенной поддержки этого в sbt-spark-packages нет, но методики из некоторых пакетов, включая spark-testing-base (https://github.com/holdenk/spark-testing-base), способны послужить в качестве фундамента для вашей собственной кросс-сборки.

Резюме

После чтения этой главы у вас должны были появиться базовые знания о библиотеках машинного обучения и потоковой обработки фреймворка Spark, а также о способах получения доступа к пакетам, созданным сообществом, в тех случаях, когда одного фреймворка Spark недостаточно. Если машинное обучение и потоковая обработка играют ключевую роль в вашей работе, то рекомендуем после прочтения данной книги обратиться к более специализированным источникам информации. Их перечень вы можете найти в разделе «Вспомогательная литература и материалы» на с. 15.

Мы надеемся, что эта книга оказалась полезна, и напоминаем: она будет великолепным праздничным подарком[1] каждому из членов вашей семьи, сослуживцам и даже вашим домашним питомцам[2].

Говоря серьезно: если вы заметили какие-либо возможности улучшить эту книгу (ошибки или неполноту информации), то призываем связаться с нами по адресу, указанному в разделе «Примечания к первому изданию» на с. 15. Теперь, после прочтения данной книги, вы должны быть готовы поднять свои знания Apache Spark на новый уровень и относительно легко справляться с масштабными задачами обработки данных. Спасибо за то, что включили нас в свое путешествие по Apache Spark. Всего хорошего, и пусть в вашем коде будет поменьше ошибок, а кластеры работают надежно.

[1] До ближайшего праздника еще далеко? Ничего страшного, она станет прекрасным подарком на день рождения и не на день рождения тоже.

[2] Мы обнаружили, что почтовая упаковка этой книги нравится домашним питомцам даже больше, чем сама книга.

Приложение. Тонкая настройка, отладка и другие аспекты, обычно игнорируемые разработчиками

Тонкая настройка фреймворка Spark и выбор размеров кластеров

Как вы помните из главы 2, объект `SparkSession` или `SparkContext` содержит конфигурацию Spark, определяющую способ запуска приложения. Большинство настроек фреймворка можно регулировать только на уровне приложения. Эти настройки оказывают сильное влияние на скорость выполнения заданий и вероятность их успешного завершения. Настройки Spark по умолчанию выбраны так, чтобы гарантировать отправку заданий на выполнение при очень маленьких размерах кластеров, и не рекомендуются для промышленной эксплуатации.

Чаще всего эти настройки приходится менять для более полного использования имеющихся ресурсов, а иногда даже для того, чтобы задание вообще хоть как-то заработало. Spark предоставляет довольно ограниченные возможности управления настройкой среды, и корректировка этих настроек часто позволяет повысить производительность задания до должного уровня. Например, в главе 6 мы поясняли, что ошибки нехватки памяти в исполнителях — частая причина сбоев заданий Spark. Хотя лучше всего сосредоточить усилия на изложенных в предыдущих главах методиках предотвращения асимметрии данных и дорогостоящих перетасовок, но уменьшение количества исполнителей с увеличением их размера тоже может помочь предотвратить сбои.

Настройка заданий Spark — в равной степени наука и искусство. Выбор настроек зависит от размера и настроек хранилища данных, размера отправляемых

на выполнение заданий (объемов обрабатываемых данных) и их вида. Например, потребности заданий, которые кэшируют много данных и совершают множество итеративных вычислений, отличаются от потребностей заданий, выполняющих небольшое количество очень крупных перетасовок. Тонкая настройка приложения также зависит от того, какие цели преследует ваша команда. В одних ситуациях (скажем, в случае разделяемых ресурсов) может понадобиться настроить задание так, чтобы оно успешно завершалось и при минимальном количестве используемых ресурсов. В других для достижения оптимальной производительности приложения может потребоваться максимально увеличить доступные ресурсы.

В данном разделе мы не стремимся всесторонне познакомить вас с настройкой и запуском приложений Spark. Вместо этого мы хотели бы скорее обрисовать ситуацию в целом и дать советы по использованию настроек, существенно влияющих на производительность. Другими словами, мы предполагаем, что у вас уже есть подходящая для запуска приложения система, но вам хотелось бы отрегулировать ее так, чтобы приложения работали быстрее или могли работать с большими объемами данных.

Корректировка настроек Spark

Объект `SparkContext` (`SparkSession` в версии 2.0 и более поздних. — *Примеч. пер.*) символизирует подключение к приложению Spark. Он включает объект `SparkConf`, определяющий настройки приложения Spark в вашей системе. А тот содержит все настройки, значения по умолчанию и информацию о среде, обуславливающие поведение приложения Spark. Эти настройки представлены в виде пар «ключ — значение»; например, установка значения свойства `spark.executor.instances` равным 5 означает отправку на выполнение задания с использованием пяти исполнителей (виртуальных машин Java Spark).

Можно создать объект `SparkConf` со всеми настройками до того, как приступить к `SparkContext`. Для некоторых из свойств, например названия приложения, имеются соответствующие вызовы API. В остальных случаях значения свойств объекта `SparkConf` устанавливаются непосредственно с помощью метода `.set()`, принимающего в качестве аргументов произвольные пары «ключ — значение». Чтобы задать различные настройки приложения Spark для каждого отправляемого на выполнение задания, можно просто создать пустой объект `SparkConf` и передать настройки во время выполнения. См. документацию Spark[1] (http://spark.apache.org/docs/latest/configuration.html#dynamically-loading-spark-properties).

Настройки работающих заданий можно найти на вкладке environment (среда) веб-интерфейса.

[1] Это хороший вариант для приложений Spark, предназначенных для работы во множестве различных сред или для подобных нашему сценариев использования, с веб-приложением, которое отправляет задания Spark на выполнение изнутри приложения.

Как выяснить актуальные характеристики своего кластера

Основные ресурсы, которыми управляет приложение Spark, — это CPU (количество ядер) и оперативная память. Запросы Spark не могут запрашивать больше ресурсов, чем доступно в среде их выполнения. Следовательно, важно понимать, сколько ядер CPU и оперативной памяти доступно в среде, где будет выполняться задание. Если вы настроили собственный кластер, то ответы на эти вопросы очевидны. Но часто приходится работать в системах, которые создаются и настраиваются не нами, так что важно уметь выяснять объем доступных ресурсов (или знать, какие вопросы задавать системному администратору). Ответы на данные вопросы зависят от типа вашей системы, но в целом есть четыре основных аспекта, касающихся нашего аппаратного обеспечения, которые нужно знать[1].

- Насколько большим может быть объем запроса? В большинстве систем есть ограничение на размер запроса, устанавливающее предел объема ресурсов, потенциально доступного каждому исполнителю и драйверу. В *кластерном режиме YARN* данное ограничение соответствует максимальному размеру контейнера YARN. Все исполнители и драйвер должны «вписываться» в это ограничение. Если говорить об оперативной памяти, то исполнители и драйвер требуют предоставленный объем памяти плюс перерасход. Мы рассмотрим вопрос расчета перерасхода памяти в подразделе «Подсчет перерасхода памяти в исполнителях и драйвере» на с. 305. В *клиентском режиме Spark* драйвер выполняется как процесс на клиенте, вследствие чего кластеру требуется выделить лишь ресурсы, необходимые для исполнителей Spark, а для драйвера — нет.
- Каков размер узлов? Чтобы определиться с количеством исполнителей и ядер CPU, выделяемых каждому из них, важно знать, сколько оперативной памяти и ресурсов CPU имеется в каждом узле, поскольку один исполнитель может использовать ресурсы только одного узла. Объем доступной каждому узлу оперативной памяти, вероятно, больше одного контейнера или равен ему. Однако этот вопрос все равно важен для определения количества исполнителей. Например, наш кластер состоит из трех узлов по 20 Гбайт каждый. Даже если ограничения контейнера YARN — 15 Гбайт, запустить четыре исполнителя мы не можем, ведь четвертый пришлось бы распределить по двум узлам.
- Сколько узлов в вашем кластере? Сколько из них работает в текущий момент? В целом оптимально, чтобы в каждом узле был хотя бы один исполнитель. Знание количества узлов может помочь выяснить и общий объем ресурсов.
- Какой процент ресурсов доступен в той системе, где задание будет передаваться на выполнение? Если кластер используется совместно, то насколько он загружен? Попадает ли передаваемое на выполнение задание в очередь и какой

[1] Всесторонний ответ на эти вопросы для приложений, использующих кластерный режим YARN, можно найти в следующей статье: http://alpinedata.com/how-to-use-the-yarn-api-to-determine-resources-available-for-spark-application-submission-part-1/.

объем ресурсов этой очереди доступен? Для периодических заданий может понадобиться запросить API YARN (или Mesos), чтобы выяснить, сколько израсходовано ресурсов, прежде чем передавать задание на выполнение. Зачастую если приложение Spark запрашивает больший объем ресурсов, чем имеется в соответствующей очереди, но не больше, чем во всем кластере, то запрос не завершается сбоем, а оказывается в состоянии ожидания. Выяснение доступных для очереди ресурсов зависит от типа задействованного в системе планировщика. При применении планировщика по вычислительным ресурсам каждому пользователю доступна фиксированная доля имеющихся ресурсов. В случае же недискриминационного планировщика работающие в данный момент приложения делят ресурсы поровну. Поведение обоих планировщиков хорошо описано в документации Spark. Мы также обрисовали в общих чертах способ получения данной информации с помощью API YARN в этом сообщении (http://alpinedata.com/how-to-use-the-yarn-api-to-determine-resources-available-for-spark-application-submission-part-3/) из блога одного из авторов.

Основные настройки Spark Core: сколько ресурсов нужно выделять приложению Spark

Объекты `SparkSession`/`SparkContext` запускают JVM для исполнителей (а в кластерном режиме YARN — и для драйвера). Напомним, что исполнители запускают задания (для вычисления всех секций) на доступных каждому из них ядрах процессора. Более того, часть ресурсов каждого из исполнителей расходуется на вычисления, а часть — на кэширование. Размер драйвера, исполнителей и количество выделенных каждому исполнителю ядер CPU задаются в конфигурации и остаются неизменными на время жизни приложения Spark. Размеры всех исполнителей должны быть одинаковыми. Когда динамического выделения памяти нет (см. подраздел «Выделение ресурсов кластера и динамическое выделение» на с. 308), количество исполнителей также не меняется. Когда же оно есть, Spark между этапами может отправлять исполнители в резерв. Хотя это уже детально обсуждалось в главе 2, не помешает напомнить: размеры всех исполнителей, драйвера и количество ядер каждого драйвера остаются фиксированными независимо от размера запроса. Так что, хотя благодаря динамическому выделению Spark может добавить дополнительный исполнитель для вычисления задания, предоставить исполнителю больше ресурсов для вычисления особенно дорогостоящей секции (или при освобождении ресурсов среды) он не может.

Прежде всего рассмотрим, что значит каждая из настроек (табл. П.1). Затем обсудим, как оптимально распределить доступные ресурсы между драйверами и исполнителями.

Полный объем памяти, необходимой всем исполнителям и драйверу (включая перерасход для обоих случаев), не может превышать объема доступной в кластере оперативной памяти. В клиентском режиме YARN драйвер не использует ресурсы кластера.

Таблица П.1

Название параметра Spark	Смысл	Значение по умолчанию	Ограничения	Рекомендации
spark.driver.memory	Размер драйвера Spark в мегабайтах	1024 Мбайт	В кластерном режиме YARN не должен превышать размер контейнера YARN, включая перерасход памяти	При отправке больших наборов RDD на драйвер или выполнении большого количества локальных вычислений может понадобиться более высокое значение
spark.executor.memory	Размер каждого из исполнителей Spark	1024 Мбайт	Один исполнитель + + перерасход памяти не может превышать ограничение для одного запроса (размер одного контейнера YARN)	Увеличение размера работников Spark способно предотвратить ошибки нехватки памяти, особенно если для задания требуется несбалансированная перетасовка, но может привести к снижению производительности
spark.executor.cores	Количество виртуальных ядер, выделяемых каждому исполнителю	1	Количество ядер, доступное в контейнере YARN	Должно быть около 5. При наличии ресурсов можно увеличить

Подсчет перерасхода памяти в исполнителях и драйвере

В кластерном и клиентском режимах YARN можно вручную задавать перерасход памяти как исполнителей, так и драйвера. В обоих режимах перерасход памяти исполнителя задается с помощью значения `spark.yarn.executor.memoryOverhead`. В кластерном режиме YARN память драйвера задается благодаря `spark.yarn.driver.memoryOverhead`, а в клиентском данное значение называется `spark.yarn.am.memoryOverhead`. В любом случае, если эти значения не заданы, перерасход памяти определяется следующим уравнением[1]:

```
перерасход памяти = Max(КОЭФФИЦИЕНТ_ПЕРЕРАСХОДА_ПАМЯТИ ×
          × запрошенный_объем_памяти, МИНИМАЛЬНЫЙ_ПЕРЕРАСХОД_ПАМЯТИ).
```

где `КОЭФФИЦИЕНТ_ПЕРЕРАСХОДА_ПАМЯТИ` = 0.10;

`МИНИМАЛЬНЫЙ_ПЕРЕРАСХОД_ПАМЯТИ` = 384 Мбайт.

[1] Подробная информация о перерасходе памяти и руководство по ее настройке: https://www.cloudera.com/documentation/enterprise/5-6-x/topics/cdh_ig_running_spark_on_yarn.html.

Какого размера должен быть драйвер Spark

В целом большую часть вычислений при выполнении запроса Spark совершают исполнители, так что повышение размера драйвера редко приводит к ускорению вычислений. Однако отправка на драйвер слишком больших объемов данных или выполнение слишком масштабных локальных вычислений может повлечь сбои заданий. Следовательно, увеличение доступной драйверу оперативной памяти и, соответственно, значения `spark.driver.maxResultSize` способно предотвратить ошибки нехватки памяти на драйвере.

Независимо от памяти драйвера, объем возвращаемых на драйвер результатов не может превышать значения spark.driver.maxResultSize. Оно ограничивает общий размер сериализованных результатов со всех секций, отправляемых на драйвер с помощью операции collect(). Данная настройка приводит к происходящим раньше и более очевидным сбоям тех заданий, которые чреваты ошибками нехватки памяти. Значение по умолчанию для этого параметра — 1g, что относительно немного. Если ваше задание требует сбора больших объемов результатов и отсутствует конкуренция за ресурсы с другими пользователями, то можно установить значение maxResultSize в 0, и Spark будет игнорировать это ограничение.

По нашему опыту, хорошее эвристическое правило — задавать минимально возможное значение этого параметра, которое бы не приводило к ошибкам нехватки памяти на драйвере, то есть предоставлять исполнителям максимальный объем ресурсов.

В кластерном режиме YARN и Mesos драйвер можно запустить с бо́льшим количеством ядер, задав значения параметра `spark.driver.cores`, и выполнять как многопоточный процесс. В противном случае для драйвера достаточно одного ядра.

Несколько больших исполнителей или много маленьких?

Мы знаем, что общий объем ресурсов, запрашиваемых исполнителями и драйвером, не может превышать доступного объема ресурсов, а отдельные исполнители не могут запрашивать больше памяти или больше ядер CPU, чем выделено для одного узла (или контейнера). Однако это оставляет множество вариантов распределения ресурсов по исполнителям Spark. Например, допустим, что передаем задание на выполнение кластеру, состоящему из узлов по 20 Гбайт и шести ядер. Имеет ли смысл создать четыре шестиядерных исполнителя по 20 Гбайт или восемь трехъядерных исполнителей по 10 Гбайт? Хороший вопрос! Мы потратили несколько месяцев, пытаясь разработать алгоритм для ответа на данный вопрос, и хотя в некоторых случаях обоснованное предположение относительно выделения ресурсов возможно, поиск оптимальной конфигурации одного приложения в одном кластере

отнюдь не точная наука. Вместо этого мы попытаемся дать несколько советов по распознанию последствий выбора слишком больших или слишком маленьких исполнителей в смысле или CPU, или памяти. Надеемся, что эти советы помогут разумно выбрать конфигурацию приложения и определить, какие изменения нужно внести в задание при обнаружении признаков неправильной настройки.

Много маленьких исполнителей

У большого количества маленьких исполнителей есть два потенциальных недостатка. Первый касается риска исчерпания ресурсов при вычислении секции, как мы обсуждали в главе 5. Поскольку одну секцию можно вычислять только на одном исполнителе, то размер каждой секции ограничивается пространством, в котором она вычисляется. Следовательно, возникает риск нехватки памяти, сброса данных на диск при необходимости перетасовки, кэширования несбалансированных данных или выполнения очень дорогостоящих узких преобразований. Если исполнителям выделяется только по одному ядру процессора, то на каждом из них можно выполнять только одну задачу, что не дает воспользоваться преимуществами таких объектов, как транслируемые переменные, которые приходится рассылать по исполнителям (а не по секциям, как другие определяемые в драйвере переменные).

Вторая проблема — при слишком большом количестве исполнителей ресурсы используются неэффективно. У каждого исполнителя имеется перерасход памяти, причем взаимодействие между исполнителями тоже требует расходов ресурсов, даже если они располагаются в одном узле. Как вы помните из обсуждения вопроса перерасхода памяти выше, минимальный его объем составляет чуть менее 400 Мбайт. Следовательно, при большом количестве исполнителей по 1 Гбайт почти 25 % памяти каждого исполнителя в нашем кластере будет перерасходоваться, а не идти на вычисления. Нам кажется, что при наличии ресурсов имеет смысл выделять исполнителям не менее чем по 4 Гбайт памяти, а дополнительную память выделять, исходя из коэффициента перерасхода памяти (10 % памяти исполнителя).

Несколько больших исполнителей

Очень большие исполнители способны приводить к неэкономному расходованию ресурсов просто из-за проблемы разбиения по интервалам[1]. Для полного использования всех ресурсов при размере драйвера меньшем, чем один узел, может понадобиться более одного исполнителя на узел. Например, допустим, что в нашем кластере есть только четыре очень больших узла, а вычисления требуют очень небольшого объема памяти драйвера. В этом случае три очень больших исполнителя и драйвер размером в половину любого из них — неэкономичное решение,

[1] Для настоящих фанатов алгоритмов поиск оптимального размера и количества исполнителей подобен NP-полной задаче о рюкзаке. Размеры (и количество ядер) исполнителей фиксированы и должны «помещаться» в самых разных узлах.

ведь половина последнего узла вообще не применяется. Более того, очень большие исполнители могут приводить к задержкам при сборе мусора, ведь при большем размере кучи обработка откладывается вплоть до запуска сборки мусора, а значит, паузы на сборку мусора могут затягиваться. Большое количество ядер для каждого исполнителя, похоже, снижает производительность из-за ограничений HDFS по обработке большого количества конкурентных потоков[1]. Сэнди Риза (Sandy Ryza) рекомендует не более пяти ядер CPU на один исполнитель. У нас встречались задания, в которых использовалось немного больше (6–7 ядер), но, похоже, назначение исполнителям более чем семи или восьми ядер по меньшей мере не повышает производительность и лишь понапрасну расходует ресурсы CPU. Это ограничение на количество ядер CPU должно коррелировать с ограничением памяти исполнителей, чтобы CPU и память расходовались на кластере более или менее равномерно. Другими словами, мы получали неплохие результаты, когда определяли количество исполнителей, исходя из ресурсов CPU — путем деления количества ядер CPU на каждом узле примерно на 5 — с последующим выделением памяти исполнителю на основе полученного значения.

Выделение ресурсов кластера и динамическое выделение

Динамическое выделение — процесс, при котором приложение Spark может запрашивать/переводить в резерв исполнители по мере необходимости, в течение жизненного цикла приложения. Оно способно резко повышать производительность, особенно в загруженных кластерах, поскольку позволяет приложению использовать ресурсы по мере появления и освобождать их, когда они более не требуются для заданий.

Для добавления или удаления исполнителей при динамическом выделении существуют специальные правила. Во-первых, Spark запрашивает дополнительные исполнители при наличии ожидающих выполнения задач. Во-вторых, он переводит в резерв те исполнители, которые не используются для вычисления заданий на протяжении определяемого параметром `spark.dynamicAllocation.executorIdleTimeout` промежутка времени (по умолчанию 60 секунд). При настройках по умолчанию Spark не переводит в резерв те исполнители, которые содержат закэшированные данные, поскольку после перевода исполнителя в резерв их придется вычислять заново. Поменять такое поведение можно, задав любое значение параметра `spark.dynamicAllocation.cachedExecutorIdleTimeout`, кроме значения по умолчанию: `infinity`. В этом случае исполнители с закэшированными данными будут переводиться в резерв после их бездействия в течение заданного промежутка времени.

---

[1] См. статью Сэнди Ризы (http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/) и данные из сообщения с сайта Stack Overflow (https://stackoverflow.com/questions/24622108/apache-spark-the-number-of-cores-vs-the-number-of-executors).

Можно настроить количество исполнителей, с которого Spark начинает при запуске приложения, с помощью параметра `spark.dynamicAllocation.initialExecutors`, по умолчанию равное нулю. Если вы заранее знаете, что приложение будет запускать дорогостоящие задания и в кластере доступны ресурсы, то лучше увеличить это значение. В противном случае предпочтительнее оставить значение по умолчанию, благодаря чему приложение сможет постепенно масштабировать ресурсы. Существуют также параметры конфигурации для минимального и максимального количества исполнителей, применяемых для вычисления заданий. Мы предложили бы во избежание захвата всех ресурсов кластера задавать максимальное значение равным объему ресурсов, доступному пользователю, который отправляет приложение на выполнение на кластер.

Но поскольку динамическое выделение не позволяет изменять размер исполнителей, то все равно придется определять эти размеры до запуска задания. Рекомендуем устанавливать такой размер исполнителей, как если бы вы пытались задействовать все ресурсы кластера. Это гарантирует правильное выделение ресурсов в случае дорогостоящих вычислений, когда Spark запросит максимальное количество исполнителей. Одно из возможных исключений — случай кластера с очень высокой пропускной способностью. В такой ситуации динамическое выделение дает возможность ускорить повторное использование ресурсов, если пространство в узлах становится доступно по частям.

Ограничения динамического выделения

Настройка динамического выделения представляет собой не совсем простую задачу. Чтобы оно заработало, необходимо выполнить следующие действия.

1. Установить значение `true` параметра `spark.dynamicAllocation.enabled`.
2. Настроить внешний сервис перетасовки на всех рабочих узлах. Это зависит от диспетчера кластера, см. подробности в документации Spark (http://spark.apache.org/docs/latest/job-scheduling.html#configuration-and-setup).
3. Установить значение `true` параметра `spark.shuffle.service.enabled`.
4. Не задавать значение параметра `spark.executor.instances`. Spark проигнорирует настройки динамического выделения, даже если оно настроено, и будет использовать количество исполнителей из этого параметра при условии, что он включен в конфигурацию.

В некоторых случаях если настройки гласят, что нужно использовать динамическое выделение, но сервис перетасовки настроен неправильно, то задание может зависнуть в состоянии ожидания, поскольку у рабочих узлов отсутствует механизм запроса исполнителей. Если вы уверены, что у кластера достаточно ресурсов, то при подобном поведении следует убедиться в правильности настроек сервиса перетасовки на всех рабочих узлах и в верном указании пути класса к этому сервису в конфигурации YARN.

Разделение места внутри исполнителя

На рис. 2.4 мы предполагали, что исполнители, по сути, представляют собой JVM, в которых одна часть места в памяти выделена для кэширования, а другая — для выполнения. Хотя это правда, на самом деле применяемая внутри исполнителя память разделяется более сложным образом, чем можно предположить, исходя из данной схемы, поскольку отдельные области не статичны. Во-первых, размер JVM, задаваемый свойством `spark.executor.memory`, не включает перерасхода памяти, так что исполнители Spark требуют больше памяти кластера, чем упомянутое значение. В пределах памяти исполнителя часть пространства резервируется для внутренних метаданных Spark и пользовательских структур данных (по умолчанию около 25 %). Оставшееся место на исполнителе, в документации Spark называемое `M`, служит для выполнения и хранения. Предназначенная для выполнения память — та, что требуется для вычисления преобразований Spark. Полный объем предназначенной для кэширования и выполнения памяти исполнителя задается параметром конфигурации `spark.memory.fraction`, означающим фиксированную долю памяти. Ошибки нехватки памяти во время преобразований или сброса на диск закэшированных секций обычно вызываются ограничениями этого общего для хранения и выполнения пространства. Размер `M` по умолчанию — 0,6, то есть 60 % памяти исполнителя отдается на хранение и выполнение. Хотя можно уменьшить используемое для хранения внутренних метаданных пространство, увеличив `M`, это довольно небезопасное действие, ведь такое значение служит для предотвращения вызываемых внутренними процессами ошибок нехватки памяти.

В пределах `spark.memory.fraction`, которую далее мы будем называть `M`, часть пространства резервируется под хранилище, а остальную можно использовать для хранения и выполнения. Под хранилищем в данном случае понимается расположенная в оперативной памяти группа секций, неважно, сериализованных или нет. Вместо того чтобы выделять для хранения фиксированное пространство, Spark разрешает приложениям, ничего не кэширующим в оперативной памяти, задействовать всю долю памяти для выполнения. В пределах этой памяти Spark определяет область под названием `R` для хранения кэшированных данных. Ее размер определяется в процентном соотношении к `M` и задается параметром `spark.memory.storageFraction`. Область `R` представляет собой место, которое Spark не высвобождает для выполнения при наличии закэшированных данных. Spark делает возможным сохранение больших, чем помещается в `R`, объемов данных, но разрешает вытеснение дополнительных секций, если это понадобится для будущих заданий[1]. Другими словами, для кэширования набора RDD без риска вытеснения каких-либо секций из памяти все закэшированные должны помещаться в `R`, размер которого определяется как:

```
R = spark.executor.memory ×
    × spark.memory.fraction × spark.memory.storageFraction.
```

[1] До Spark 1.6.0 размеры памяти хранения и выполнения четко разделялись значением параметра spark.memory.storageFraction.

В следующей схеме мы попытаемся проиллюстрировать соотношение между M и R. Каждый прямоугольник соответствует одному исполнителю Spark. Светло-серый сектор внизу представляет собой R. Пространство под перерасходом памяти — M.

На рис. П.1 приводится вариант кэширования двух различных наборов RDD. Показанные секции — те, что кэшированы на данном конкретном исполнителе. Отмеченная темно-серым цветом секция — использованная наиболее давно, независимо от очередности секционирования. А поскольку секции не занимают все пространство в предназначенной для хранения доле памяти R, то масштабные вычисления (представленные на рисунке «взрывом») могут задействовать место в предназначенной для хранения доле памяти.

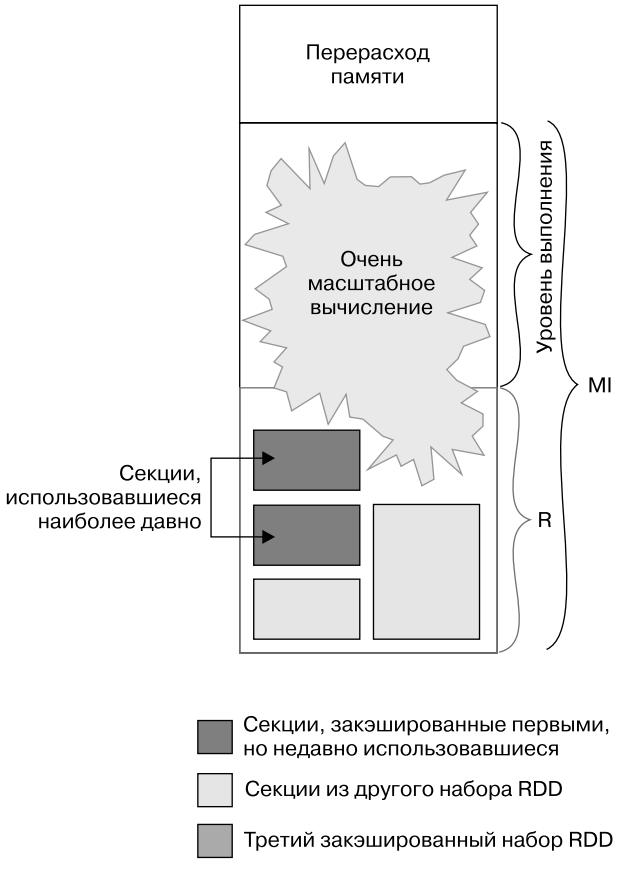

Рис. П.1. Для одного-единственного вычисления может потребоваться все пространство, доступное в доле памяти M

Далее предположим: то же приложение включает задание, которое закэшировало еще одну секцию. Теперь все закэшированные секции вместе занимают больше

места, чем R (как видите, зеленые квадратики заходят за красную линию). Это допустимо, поскольку ни для одного вычисления данное пространство не требуется. Представим также, что в задании применялась темно-серая секция; это делает светло-серую секцию использовавшейся наиболее давно. Результат показан на рис. П.2.

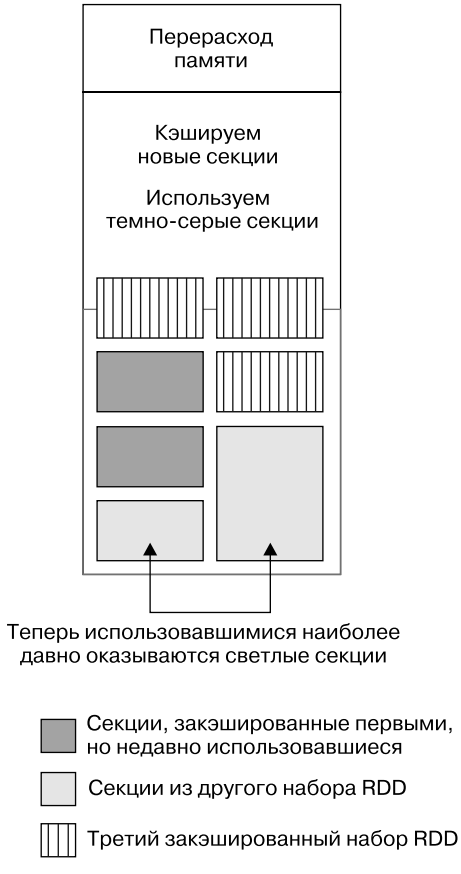

Рис. П.2. Закэшированные секции могут выходить за пределы пространства, выделенного в соответствии с параметром spark.memory.storageFraction, если ни одно вычисление их не вытесняет

Теперь допустим следующее: мы выполняем очень масштабное вычисление на этом исполнителе, приводящее к рис. П.3. В силу того что закэшированные данные занимают больше места, чем имеется в области R, лишние секции вытесняются, чтобы освободить память для вычисления. Это происходит с теми секциями, которые применялись наиболее давно (светло-серые), поскольку фреймворк Spark задействует кэширование типа *«использовавшиеся наиболее давно»* (least recently used, LRU). Дополнительную информацию о LRU-кэшировании см. в одноименном подразделе на с. 145.

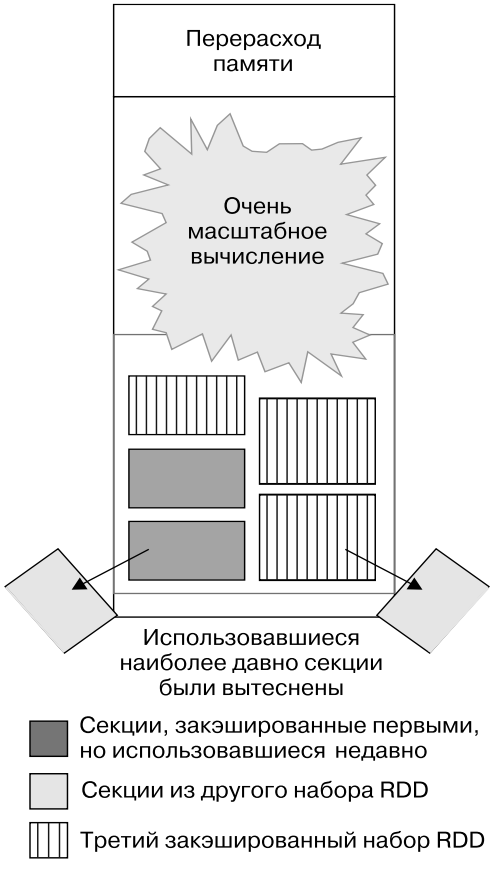

Рис. П.3. Большое вычисление может привести к вытеснению закэшированных секций, если предназначенная для хранения доля памяти переполнена. Сначала вытесняются секции, использовавшиеся наиболее давно

Корректировка настроек памяти и доли памяти для хранения сильно зависит от выполняемых вычислений. В случае некэшируемых данных эти настройки не имеют особого значения, ведь вся область `M` все равно используется для вычислений. Однако если приложению требуется неоднократный доступ к набору RDD, а его кэширование в памяти повышает производительность, то может иметь смысл увеличить размер предназначенной для хранения доли памяти, чтобы предотвратить вытеснение тех наборов RDD, которые необходимы в кэше.

Один из способов прочувствовать размеры набора RDD — написать программу, которая бы кэшировала его. После этого необходимо запустить задание и во время его работы заглянуть на страницу Storage (Хранилище) веб-интерфейса.

На рис. П.4 мы видим, как закэшированные объект `DataFrame` и набор RDD отображаются в веб-интерфейсе. Столбец Size in memory (Размеры в памяти) показывает размер соответствующей структуры данных в оперативной памяти.

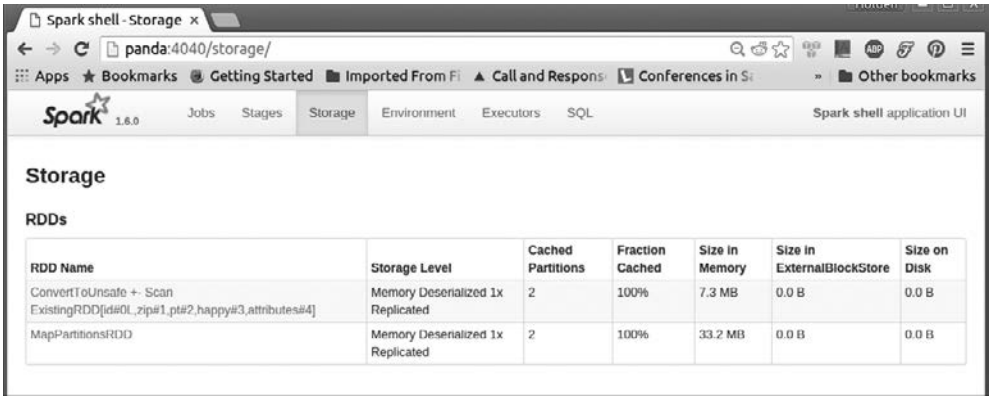

Рис. П.4. Вкладка Storage веб-интерфейса

Количество и размер секций

Как мы обсуждали в главе 6, в Spark отсутствует механизм установки оптимального количества используемых секций. По умолчанию при создании набора RDD путем чтения из устойчивого хранилища количество секций соответствует количеству фрагментов, задаваемых входным форматом (обычно это количество частей файла MapReduce). Можно явным образом менять количество секций с помощью операций `coalesce`, `repartition` или во время широких преобразований, таких как `reduceByKey` или `sort`. Если количество секций для широкого преобразования не задано, Spark по умолчанию использует количество, задаваемое параметром конфигурации `spark.default.parallelism`.

Значение по умолчанию параметра spark.default.parallelism зависит от среды выполнения приложения. В кластерном режиме YARN его значение составляет количество ядер × количество исполнителей (другими словами, максимальное количество одновременно выполняемых задач). Это минимальное количество секций, но оно не обязательно будет оптимальным.

Так сколько же секций следует использовать при широком преобразовании или в качестве значения параметра `spark.default.parallelism`?

Как и с большинством наших советов в данном приложении, однозначного ответа на этот вопрос не существует. Вообще говоря, увеличение количества секций по-

вышает производительность вплоть до определенного значения, после которого оно приводит к чрезмерному перерасходу ресурсов. Как минимум количество секций должно быть равно количеству ядер, поскольку меньшее значение приведет к простою CPU. Увеличение количества секций также снижает количество ошибок нехватки памяти, так как приводит к работе каждого исполнителя Spark с меньшим подмножеством данных. Одной из методик в этом случае может быть определение максимального размера секции, при котором она все еще «помещается» в выделенной для одной задачи памяти, и расчет на его основе необходимого количества секций (частей, на которые должен быть разбит набор RDD), при котором ни одна секция не превышает этого размера. Как вы помните из главы 2, каждый исполнитель может выполнять одну задачу на каждом ядре, а одна секция соответствует одной задаче. Как мы объясняли в подразделе «Разделение места внутри исполнителя» на с. 310, размер памяти для вычислений, доступной исполнителю Spark, находится в диапазоне между `M` и `M-R` в зависимости от объема закэшированных данных. Например:

```
память_для_вычислений (M) <
  (spark.executor.memory - перерасход_памяти) × spark.memory.fraction
```

А в случае наличия закэшированных данных:

```
память_для_вычислений (M - R) <
    (spark.executor.memory - перерасход_памяти) ×
    × spark.memory.fraction ×
    × (1 - spark.memory.storage.fraction)
```

В предположении равномерного распределения памяти между задачами каждая задача должна использовать не более:

```
память_на_задание =
    = память_для_вычислений / spark.executor.cores
```

Следовательно, число секций должно быть:

```
число_секций =
    = размеры_этапа_перетасовки / память_на_задание
```

Если секции превышают этот размер, то Spark не сможет производить вычисления такого количества задач одновременно и ресурсы CPU будут расходоваться впустую, поскольку одна задача может использовать только одно ядро CPU. Кроме того, может повыситься вероятность возникновения ошибок нехватки памяти.

Оценить размеры этапа можно с помощью некой имеющейся в веб-интерфейсе информации. Во-первых, если закэшировать набор RDD в оперативной памяти и посмотреть его размер в веб-интерфейсе, то можно ожидать, что вычисления потребуют как минимум такой же объем памяти, ведь, предположительно, вычисление требует сначала загрузки данных. Можно также попробовать понаблюдать за этапом перетасовки с целью определить, каковы затраты памяти на нее. Если мы обнаружим, что во время этапа перетасовки данные на диск не сбрасываются,

то можно сделать вывод: все секции отлично помещаются в памяти и не нужно увеличивать их количество.

Вы увидите в веб-интерфейсе, сбрасывают ли задачи данные на диск и сколько именно. Чтобы исследовать какой-либо из этапов, перейдите на вкладку jobs (задания) веб-интерфейса во время работы задания. Затем щелкните на выполняемом этапе. Там вы сможете увидеть все подробности о нем. В них перечисляются метрики производимой работы для каждого исполнителя и выполняемых задач. В таблице задач два последних столбца содержат размер сбрасываемых во время перетасовки данных: десериализованных в памяти и сериализованных на диске. Если в этих столбцах стоят нули, то ни одна из задач не сбрасывала данные на диск[1].

Если задание сбрасывает данные на диск, то имеет смысл оценить объем перетасовки и попытаться поточнее задать количество секций. Конечно, «объем перетасовки» — не та информация, которую можно определить точно. Сэнди Риза предлагает использовать соотношение между объемом сбрасываемых данных в памяти (показанных в веб-интерфейсе в столбце Shuffle spill (memory)) и на диске (показанных там же в столбце Shuffle spill (disk)), умножая его на размер данных на диске для получения приближенного размера файлов перетасовки[2]. Подробнее на этой процедуре мы остановимся в следующем разделе.

Объем сбрасываемых данных (в памяти) — размер пространства, занимаемого записями в памяти *до* сброса на диск. Объем сбрасываемых данных (на диске) — размер пространства, занимаемого записями на диске *после* их сброса. Следовательно, соотношение между объемом сбрасываемых данных в памяти и на диске — мера того, насколько больше места записи занимают в памяти по сравнению с диском. Мы бы назвали это коэффициентом увеличения объема в памяти. Формализовать его можно с помощью такого уравнения:

коэффициент увеличения объема в памяти = объем сбрасываемых данных (в памяти) / объем сбрасываемых данных (на диске).

В веб-интерфейсе на вкладке Initial Stages (Начальные этапы) можно видеть общий размер записываемых при перетасовке данных. Он соответствует размеру перетасовочных файлов, записанных на диск.

---

[1] См. http://jason4zhu.blogspot.com/2015/06/roaming-through-spark-ui-and-tune-performance-upon-a-specific-use-case.html.

[2] Речь идет о часто цитируемом сообщении из блога Ризы (http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/), посвященном тонкой настройке Spark и параллелизму. Оно было написано в расчете на более старую версию Spark, поэтому вы можете заметить, что его описание управления памятью отличается от нашего. Однако большая часть информации, в частности обсуждение вопроса установки размеров секций, все еще вполне адекватна.

Следовательно, размер этих файлов в памяти равен:

размер перетасовки в памяти = размер записываемых при перетасовке данных × × объем сбрасываемых данных (в памяти) / объем сбрасываемых данных (на диске).

Опять же, несмотря на всю сложность данного метода, он представляет собой всего лишь эмпирическое правило. В нем используется некорректное допущение о том, что размеры всех записей увеличиваются в одинаковой пропорции при чтении в память. Нет никаких других вариантов решения этой проблемы, кроме простого увеличения количества секций вплоть до прекращения улучшения производительности.

Варианты сериализации

По умолчанию фреймворк Spark задействует механизм сериализации языка Java для наборов RDD и сериализацию на основе Tungsten для наборов `DataFrame`/`Dataset`. Использование объектов `DataFrame` или `Dataset` (в тех случаях, когда оно возможно) обеспечивает намного более эффективный слой сериализации, но при работе с наборами RDD доступен вариант сериализации с помощью фреймворка Kryo.

Kryo

Подобно сериализатору Tungsten, при сериализации с помощью Kryo нативно поддерживаются не все из типов, реализующих Java-интерфейс `Serializable`. Подробности взаимодействия Kryo с фреймворком Spark описаны в *Learning Spark* (кроме того, можно найти и другие примеры в свободном доступе (https://github.com/holdenk/learning-spark-examples/blob/master/src/main/java/com/oreilly/learningsparkexamples/java/BasicAvgWithKryo.java)). Можно было бы подумать, что с появлением Tungsten работа над интеграцией Kryo в Spark прекратится, но на самом деле усовершенствование сериализации на основе Kryo в Spark продолжается. В следующей версии Spark планируется добавить поддержку небезопасного сериализатора Kryo, который, возможно, будет работать даже быстрее, чем Tungsten (его можно активизировать, установив в `true` значение параметра `spark.kryo.unsafe`).

Настройки Spark. Резюме

Правильная настройка Spark может весьма значительно повысить производительность. Однако она занимает много времени и в некоторых случаях требует многих часов тестирования. Никакая тонкая настройка не позволила бы успешно завершить на объеме данных в миллиард строк несбалансированную перетасовку из раздела «Златовласка. Версия 1: решение с использованием функции groupByKey» на с. 157 (поверьте, мы пробовали). Из-за очень большого количества переменных,

участвующих в настройке приложения (размерностей кластера, трафика, размера входных данных, типа выполняемых вычислений), поиск оптимальной конфигурации Spark возможен лишь методом проб и ошибок. Однако чем лучше вы понимаете, какую информацию необходимо искать в веб-интерфейсе (например, информацию о том, сбрасываются ли данные на диск при перетасовках), тем легче проходит этот процесс. А важнее всего: хорошее понимание своей системы и того, что требуется для вычислений, помогает планировать оптимальную стратегию отправки приложения на выполнение.

Дополнительные методики отладки

Отладка — важная часть жизненного цикла создания программного обеспечения, а у отладки с помощью Spark есть свои особенности. Наиболее очевидная из ее сложностей — то, что в распределенной системе трудно определить, какая (-ие) машина (-ы) сгенерировала (-и) ошибку, а получить доступ к рабочим узлам для отладки может оказаться невозможным. Кроме того, использование в Spark отложенных вычислений способно запутать привыкших к более традиционным системам разработчиков, ведь журналы и трассы вызовов в стеке будут, на первый взгляд, наводить на ложный след.

Отладка приложения, использующего отложенные вычисления, требует отказа от допущения о непременной непосредственной связи ошибки со строкой кода, которая, как представляется, вызвала ее генерацию, или любой из непосредственно вызываемых в ней функций. Находясь в интерактивной среде разработки, можно быстро отыскать причину встреченной ошибки, добавив вызов `take(1)` или `count` для родительских наборов RDD или `DataFrame`. Однако при отладке кода готового приложения роскошь вставки `take(1)` или `count` может оказаться недоступной, в частности, из-за значительного увеличения при этом времени вычислений. Вместо этого в подобных случаях важно научиться различать два кажущихся похожими варианта отказов. Рассмотрим примеры П.1 и П.2, а также соответствующие трассы вызовов в примерах П.3 и П.4.

Пример П.1. Генерация исключения из внутреннего набора RDD

```
val data = sc.parallelize(List(1, 2, 3))
// Сгенерирует исключение при принудительной инициации вычислений
val transform1 = data.map(x => x/0)
val transform2 = transform1.map(x => x + 1)
transform2.collect() // Инициирует вычисление
```

Пример П.2. Генерация исключения из самого верхнего набора RDD

```
val data = sc.parallelize(List(1, 2, 3))
val transform1 = data.map(x => x + 1)
// Сгенерирует исключение при принудительной инициации вычислений
val transform2 = transform1.map(x => x/0)
transform2.collect() // Инициирует вычисление
```

Пример П.3. Сбой во внутреннем наборе RDD

```
17/01/23 12:41:36 ERROR Executor: Exception in task 0.0 in stage 0.0 (TID 0)
java.lang.ArithmeticException: / by zero
    at com.highperformancespark.examples.errors.Throws$$anonfun$1
    .apply$mcII$sp(throws.scala:9)
    at com.highperformancespark.examples.errors.Throws$$anonfun$1
    .apply(throws.scala:9)
    at com.highperformancespark.examples.errors.Throws$$anonfun$1
    .apply(throws.scala:9)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$class.foreach(Iterator.scala:750)
    at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
    at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
    at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
    at scala.collection.AbstractIterator.to(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
    at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
    at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13.
    apply(RDD.scala:935)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13.
    apply(RDD.scala:935)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:99)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/01/23 12:41:36 WARN TaskSetManager: Lost task 0.0
in stage 0.0 (TID 0, localhost, executor driver):
java.lang.ArithmeticException: / by zero
    at com.highperformancespark.examples.errors.Throws$$anonfun$1
    .apply$mcII$sp(throws.scala:9)
    at com.highperformancespark.examples.errors.Throws$$anonfun$1
    .apply(throws.scala:9)
    at com.highperformancespark.examples.errors.Throws$$anonfun$1
    .apply(throws.scala:9)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$class.foreach(Iterator.scala:750)
    at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
    at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
    at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
```

```
    at scala.collection.AbstractIterator.to(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
    at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
    at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:99)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/01/23 12:41:36 ERROR TaskSetManager:
Task 0 in stage 0.0 failed 1 times; aborting job
org.apache.spark.SparkException: Job aborted due to stage failure:
Task 0 in stage 0.0 failed 1 times, most recent failure:
Lost task 0.0 in stage 0.0 (TID 0, localhost, executor driver):
java.lang.ArithmeticException: / by zero
    at com.highperformancespark.examples.errors.Throws$$anonfun$1.
    apply$mcII$sp(throws.scala:9)
    at com.highperformancespark.examples.errors.Throws$$anonfun$1.apply(throws.
scala:9)
    at com.highperformancespark.examples.errors.Throws$$anonfun$1.apply(throws.
scala:9)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$class.foreach(Iterator.scala:750)
    at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
    at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
    at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
    at scala.collection.AbstractIterator.to(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
    at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
    at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
```

```
    .apply(SparkContext.scala:1944)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:99)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

Driver stacktrace:
  at org.apache.spark.scheduler.
  DAGScheduler.org$apache$spark$scheduler$DAGScheduler$$
  failJobAndIndependentStages(DAGScheduler.scala:1435)
  at org.apache.spark.scheduler.DAGScheduler$$anonfun$abortStage$1
  .apply(DAGScheduler.scala:1423)
  at org.apache.spark.scheduler.DAGScheduler$$anonfun$abortStage$1
  .apply(DAGScheduler.scala:1422)
  at scala.collection.mutable.ResizableArray$class.foreach(ResizableArray.scala:59)
  at scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:48)
  at org.apache.spark.scheduler.DAGScheduler.abortStage(DAGScheduler.scala:1422)
  at org.apache.spark.scheduler
  .DAGScheduler$$anonfun$handleTaskSetFailed$1.apply(DAGScheduler.scala:802)
  at org.apache.spark.scheduler
  .DAGScheduler$$anonfun$handleTaskSetFailed$1.apply(DAGScheduler.scala:802)
  at scala.Option.foreach(Option.scala:257)
  at org.apache.spark.scheduler.DAGScheduler.
  handleTaskSetFailed(DAGScheduler.scala:802)
  at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.
  doOnReceive(DAGScheduler.scala:1650)
  at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop
  .onReceive(DAGScheduler.scala:1605)
  at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop
  .onReceive(DAGScheduler.scala:1594)
  at org.apache.spark.util.EventLoop$$anon$1.run(EventLoop.scala:48)
  at org.apache.spark.scheduler.DAGScheduler.runJob(DAGScheduler.scala:628)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1918)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1931)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1944)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1958)
  at org.apache.spark.rdd.RDD$$anonfun$collect$1.apply(RDD.scala:935)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
  at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
  at org.apache.spark.rdd.RDD.collect(RDD.scala:934)
  at com.highperformancespark.examples.errors.Throws$.throwInner(throws.scala:11)
  ... 43 elided
Caused by: java.lang.ArithmeticException: / by zero
  at com.highperformancespark.examples.errors.Throws$$anonfun$1
  .apply$mcII$sp(throws.scala:9)
  at com.highperformancespark.examples.errors.Throws$$anonfun$1.apply(throws.scala:9)
  at com.highperformancespark.examples.errors.Throws$$anonfun$1.apply(throws.scala:9)
  at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
```

```
  at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
  at scala.collection.Iterator$class.foreach(Iterator.scala:750)
  at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
  at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
  at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
  at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
  at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
  at scala.collection.AbstractIterator.to(Iterator.scala:1202)
  at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
  at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
  at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
  at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
  at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13.apply(RDD.scala:935)
  at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13.apply(RDD.scala:935)
  at org.apache.spark.SparkContext$$anonfun$runJob$5.apply(SparkContext.scala:1944)
  at org.apache.spark.SparkContext$$anonfun$runJob$5.apply(SparkContext.scala:1944)
  at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
  at org.apache.spark.scheduler.Task.run(Task.scala:99)
  at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
  ... 1 more
```

Пример П.4. Генерация «внешнего» исключения

```
17/01/23 12:45:27 ERROR Executor: Exception in task 0.0 in stage 1.0 (TID 1)
java.lang.ArithmeticException: / by zero
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply$mcII$sp(throws.scala:17)
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply(throws.scala:17)
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply(throws.scala:17)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$class.foreach(Iterator.scala:750)
    at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
    at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
    at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
    at scala.collection.AbstractIterator.to(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
    at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
    at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
```

```
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:99)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/01/23 12:45:27 WARN TaskSetManager: Lost task 0.0 in stage 1.0
(TID 1, localhost, executor driver):
java.lang.ArithmeticException: /
by zero
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply$mcII$sp(throws.scala:17)
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply(throws.scala:17)
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply(throws.scala:17)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$class.foreach(Iterator.scala:750)
    at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
    at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
    at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
    at scala.collection.AbstractIterator.to(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
    at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
    at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:99)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

17/01/23 12:45:27 ERROR TaskSetManager: Task 0 in stage 1.0 failed 1 times;
aborting job
org.apache.spark.SparkException: Job aborted due to stage failure:
Task 0 in stage 1.0 failed 1 times, most recent failure:
Lost task 0.0 in stage 1.0 (TID 1, localhost, executor driver):
java.lang.ArithmeticException: / by zero
```

```
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply$mcII$sp(throws.scala:17)
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply(throws.scala:17)
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply(throws.scala:17)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$class.foreach(Iterator.scala:750)
    at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
    at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
    at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
    at scala.collection.AbstractIterator.to(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
    at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
    at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:99)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

Driver stacktrace:
  at org.apache.spark.scheduler
  .DAGScheduler.org$apache$spark$scheduler$DAGScheduler$$failJobAndIndependentStages
  (DAGScheduler.scala:1435)
  at org.apache.spark.scheduler.DAGScheduler$$anonfun$abortStage$1
  .apply(DAGScheduler.scala:1423)
  at org.apache.spark.scheduler.DAGScheduler$$anonfun$abortStage$1
  .apply(DAGScheduler.scala:1422)
  at scala.collection.mutable.ResizableArray$class.foreach(ResizableArray.scala:59)
  at scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:48)
  at org.apache.spark.scheduler.DAGScheduler.abortStage(DAGScheduler.scala:1422)
  at org.apache.spark.scheduler.DAGScheduler$$anonfun$handleTaskSetFailed$1
  .apply(DAGScheduler.scala:802)
  at org.apache.spark.scheduler.DAGScheduler$$anonfun$handleTaskSetFailed$1
  .apply(DAGScheduler.scala:802)
  at scala.Option.foreach(Option.scala:257)
  at org.apache.spark.scheduler.DAGScheduler
  .handleTaskSetFailed(DAGScheduler.scala:802)
  at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop
```

```
  .doOnReceive(DAGScheduler.scala:1650)
  at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop
  .onReceive(DAGScheduler.scala:1605)
  at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop
  .onReceive(DAGScheduler.scala:1594)
  at org.apache.spark.util.EventLoop$$anon$1.run(EventLoop.scala:48)
  at org.apache.spark.scheduler.DAGScheduler.runJob(DAGScheduler.scala:628)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1918)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1931)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1944)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1958)
  at org.apache.spark.rdd.RDD$$anonfun$collect$1.apply(RDD.scala:935)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
  at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
  at org.apache.spark.rdd.RDD.collect(RDD.scala:934)
  at com.highperformancespark.examples.errors.Throws$.throwOuter(throws.scala:18)
  ... 43 elided
Caused by: java.lang.ArithmeticException: / by zero
  at com.highperformancespark.examples.errors
  .Throws$$anonfun$4.apply$mcII$sp(throws.scala:17)
  at com.highperformancespark.examples.errors
  .Throws$$anonfun$4.apply(throws.scala:17)
  at com.highperformancespark.examples.errors
  .Throws$$anonfun$4.apply(throws.scala:17)
  at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
  at scala.collection.Iterator$class.foreach(Iterator.scala:750)
  at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
  at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
  at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
  at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
  at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
  at scala.collection.AbstractIterator.to(Iterator.scala:1202)
  at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
  at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
  at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
  at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
  at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13.apply(RDD.scala:935)
  at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13.apply(RDD.scala:935)
  at org.apache.spark.SparkContext$$anonfun$runJob$5.apply(SparkContext.scala:1944)
  at org.apache.spark.SparkContext$$anonfun$runJob$5.apply(SparkContext.scala:1944)
  at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
  at org.apache.spark.scheduler.Task.run(Task.scala:99)
  at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
  ... 1 more
```

Две приведенные трассы вызовов содержат массу информации, но большая ее часть бесполезна при поиске причины нашей ошибки. Поскольку источник ошибки — внутри работника, то можно по большей части проигнорировать информацию о трассе вызовов драйвера и исследовать вместо этого исключения, указанные ниже `17/01/23 12:41:36 ERROR Executor: Exception in task 0.0 in stage 0.0 (TID 0)`.

Частая ошибка — изучение трассы вызовов драйвера при сообщении об ошибке от исполнителя. В этом случае может казаться, что вы ищете ошибку в правильном направлении, в то время как ее первопричина находится совсем в другом месте (как в обоих предыдущих примерах).

При изучении сообщения об ошибке от исполнителя можно увидеть номер строки, соответствующий сбойной функции. Обычно остальная часть исключения не нужна для отладки, разве что речь идет о работе с операцией `mapPartitions` и возврате пользовательских итераторов, поскольку формирование цепочки итераторов происходит внутри Spark.

Об исключении сообщается дважды (один раз как предупреждение, а второй — как ошибка), поскольку Spark повторяет обработку секции при сбое. Если сбойных секций было много, то сообщение об ошибке вы получите гораздо больше двух раз.

Если выполнение приложения не было аварийно завершено, то эти исключения можно также увидеть в журналах веб-интерфейса Spark.

Определить, какой оператор привел к генерации ошибки, может быть непросто, особенно в случае анонимных внутренних функций. Если поменять примеры П.1 и П.2 так, чтобы использовались явные имена функций (допустим, как в примерах П.5 и П.6), то выяснить произошедшее будет гораздо легче. Полученная трасса вызовов теперь содержит имена интересующих нас функций (например, `at com.high-performancespark.examples.errors.Throws$.divZero(throws.scala:26)`).

Пример П.5. Переделанные вспомогательные функции

```
def add1(x: Int): Int = {
  x + 1
}
def divZero(x: Int): Int = {
  x / 0
}
```

Пример П.6. Образцы генерации исключений, переделанные для использования вспомогательных функций

```
def throwInner2(sc: SparkContext) = {
  val data = sc.parallelize(List(1, 2, 3))
  // Сгенерирует исключение при принудительной инициации вычислений
  val transform1 = data.map(divZero)
  val transform2 = transform1.map(add1)
  transform2.collect() // Инициирует вычисление
}

def throwOuter2(sc: SparkContext) = {
  val data = sc.parallelize(List(1, 2, 3))
  val transform1 = data.map(add1)
```

```
  // Сгенерирует исключение при принудительной инициации вычислений
  val transform2 = transform1.map(divZero)
  transform2.collect() // Инициирует вычисление
}
```

> Вы могли заметить, что, даже если исходной причиной ошибки является деление на ноль (исключение java.lang.ArithmeticException), на более высоком уровне исключение обернуто в объект org.apache.spark.SparkException. Для доступа к исходному исключению можно воспользоваться методом getCause.

Не все исключения обертываются в объект `org.apache.spark.SparkException`. При попытке вычисления набора RDD, данные для которого содержатся в несуществующем источнике Hadoop, трасса вызовов будет гораздо проще (пример П.7) и возвратится непосредственно исходное исключение.

Пример П.7. Исключение в случае попытки загрузки несуществующего источника данных

```
org.apache.hadoop.mapred.InvalidInputException:
Input path does not exist: file:/doesnotexist.txt
  at org.apache.hadoop.mapred.FileInputFormat.
  singleThreadedListStatus(FileInputFormat.java:285)
  at org.apache.hadoop.mapred.FileInputFormat.listStatus(FileInputFormat.java:228)
  at org.apache.hadoop.mapred.FileInputFormat.getSplits(FileInputFormat.java:313)
  at org.apache.spark.rdd.HadoopRDD.getPartitions(HadoopRDD.scala:202)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:252)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:250)
  at scala.Option.getOrElse(Option.scala:121)
  at org.apache.spark.rdd.RDD.partitions(RDD.scala:250)
  at org.apache.spark.rdd.MapPartitionsRDD.getPartitions(MapPartitionsRDD.scala:35)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:252)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:250)
  at scala.Option.getOrElse(Option.scala:121)
  at org.apache.spark.rdd.RDD.partitions(RDD.scala:250)
  at org.apache.spark.rdd.MapPartitionsRDD.getPartitions(MapPartitionsRDD.scala:35)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:252)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:250)
  at scala.Option.getOrElse(Option.scala:121)
  at org.apache.spark.rdd.RDD.partitions(RDD.scala:250)
  at org.apache.spark.rdd.MapPartitionsRDD.getPartitions(MapPartitionsRDD.scala:35)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:252)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:250)
  at scala.Option.getOrElse(Option.scala:121)
  at org.apache.spark.rdd.RDD.partitions(RDD.scala:250)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1958)
  at org.apache.spark.rdd.RDD$$anonfun$collect$1.apply(RDD.scala:935)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
  at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
  at org.apache.spark.rdd.RDD.collect(RDD.scala:934)
  at com.highperformancespark.examples.errors.Throws$
  .nonExistentInput(throws.scala:47)
  ... 43 elided
```

Отладка исключений из-за нехватки памяти драйвера может оказаться сложной задачей, особенно выяснение того, какая именно операция вызвала сбой. Хотя вам уже, наверное, надоели операторы `collect`, важно помнить, что другие операции (например, `countByKey`) способны возвращать в драйверную программу потенциально неограниченные объемы данных. В библиотеках Spark ML и MLlib для некоторых моделей требуется возвращать в драйверную программу большие объемы данных — в этом случае простейшим решением будет воспользоваться другой моделью.

Предсказать, сколько памяти понадобится драйверной программе, или выяснить источник расхода большого количества памяти может быть непросто. Более того, процесс определения объема памяти, выделяемой драйверной программе, зависит от способа запуска приложения Spark. При запуске приложения с помощью сценария `spark-submit` в «клиентском режиме» (или основанных на нем сценариев, таких как `spark-shell` или `pyspark`) JVM драйвера запускается *до* синтаксического разбора файла `spark-defaults.conf` и до создания объекта `SparkContext`. В этом случае задавать объем памяти драйвера приходится в `spark-env.sh` или прибегнув к параметру `--driver-memory` команды `spark-submit`.

При запуске Spark из Python без помощи spark-submit или любых вспомогательных средств для определения размера JVM драйвера или каких-либо настроек виртуальной машины на стороне драйвера объект конфигурации не будет учитываться, хотя формально JVM еще не была запущена. Так происходит из-за того, что JVM драйвера «за кулисами» запускается с помощью обычного сценария spark-submit. Взамен при необходимости можно задать значение переменной среды командной оболочки PYSPARK_SUBMIT_ARGS (по умолчанию равной pyspark-shell).

Обычно не включаемое в список традиционных методов отладки устранение отстающих задач или других вариантов несбалансированного секционирования тесно связано с ними и обсуждается в главе 5.

Ошибки нехватки места на диске

Ошибки нехватки места на диске могут оказаться неожиданными, но часто встречаются в кластерах с небольшим объемом дискового пространства. Иногда такие ошибки происходят из-за длительной работы командной оболочки, где в глобальной области видимости создаются наборы RDD, для которых никогда не выполняется сборка мусора. Spark записывает результаты операций перетасовки в файлы на дисках работников в своем локальном каталоге. Данные файлы удаляются только при сборке мусора для соответствующих наборов RDD; это происходит достаточно редко при большом объеме памяти, выделенной драйверной программе. Одно из решений — явным образом инициировать сборку мусора (конечно, при условии, что

наборы RDD уже вышли из области видимости); если DAG становится слишком длинным, то создание контрольных точек может обеспечить возможность сборки мусора для набора RDD.

Журналирование

Журналирование — важная составляющая отладки приложения, так что в распределенной системе полагаться на `println`, пожалуй, не стоит. Внутри Spark используется механизм журналирования `log4j` через `sl4j`, поэтому `log4j` выступит подходящим механизмом журналирования и для наших приложений, ведь мы в какой-то мере будем уверены в том, что он уже настроен.

В ранних (до 2.0) версиях фреймворка Spark API журналирования, построенный на основе библиотеки log4j, был сделан доступным для использования. В версии 2.0 и более поздних его сделали закрытым, да и в нем нет почти никакой функциональности, которая не была бы доступна непосредственно в log4j, так что лучше напрямую применять механизм журналирования log4j. Сама же автор признает себя виновной в обращении к внутренним API журналирования[1].

Можно получить доступ к той же функциональности, что и применяемый внутри Spark механизм журналирования, с помощью типобезопасного пакета `scalalogging`. В нем есть типаж `LazyLogging`, который использует макрос для замены вызова `logger.debug(xyz)` на эквивалентный, но находится за охраняющим выражением, проверяющим уровень журналирования. Простой пример журналирования для отладки: выяснение того, какие элементы были отфильтрованы. Можно добавить журналирование в пример 5.16, в результате чего получим пример П.8. Кроме того, необходимо включить в сборку библиотеку журналирования (пример П.9).

Пример П.8. Журналирование трансляции объекта HashSet некорректных местоположений панд с целью их последующей фильтрации

```
val invalid = HashSet() ++ invalidPandas
val invalidBroadcast = sc.broadcast(invalid)
def keepPanda(pandaId: Long) = {
  if (invalidBroadcast.value.contains(pandaId)) {
    logger.debug(s"Invalid panda ${pandaId} discovered")
    false
  } else {
    true
  }
}
input.filter{panda => keepPanda(panda.id)}
```

Пример П.9. Добавление библиотеки scalalogging в сборку

```
"com.typesafe.scala-logging" %% "scala-logging" % "3.5.0",
```

[1] И, написав это предупреждение, она покаялась в своих грехах.

Иногда может пригодиться подробная информация в журнале о наборе RDD/номере секции/номере попытки обработки. В этом случае для ее получения вы можете заглянуть в объект TaskContext (как в JVM, так и в Python).

Настройка журналирования

Spark использует для журналирования JVM библиотеку `log4j`, и даже в Python или R значительная часть информации журналирования генерируется из JVM. При запуске интерактивной командной оболочки одно из первых сообщений содержит инструкции по настройке уровня журналирования в интерактивном режиме:

```
To adjust logging level use sc.setLogLevel(newLevel).
For SparkR, use setLogLevel(newLevel).
```

Помимо установки уровня журналирования с помощью SparkContext, можно приспособить для настройки уровня журналирования и выводимой при журналировании информации файл `conf/log4j.properties.template`. Просто скопируйте его в `conf/log4j.properties` и поменяйте все нужные свойства; так, если Spark выводит слишком много подробностей, поменяйте уровень основного журналирования на `ERROR`, как в примере П.10.

Пример П.10. log4j.properties

```
log4j.logger.org.apache.spark.repl.Main=ERROR
```

Можно настроить разные уровни журналирования для различных механизмов журналирования (например, при настройках conf/log4j.properties.template по умолчанию уровень журналирования Spark будет WARN, а Parquet — ERROR). Это особенно удобно в случае, когда ваше приложение также использует журналирование log4j.

Если возможности настроек `log4j.properties` вас не удовлетворяют, то можно создать пользовательский файл `log4j.xml` и установить его на исполнители (включив его в JAR или с помощью параметра `--files`), добавив затем `-Dlog4j.configuration=log4j.xml` в `spark.executor.extraJavaOptions`, чтобы исполнители его «увидели».

Доступ к журналам

В период активности приложения можно легко просматривать журналы разных работников с помощью веб-интерфейса Spark. Возможности просмотра журналов после окончания работы приложения будут зависеть от используемого механизма развертывания.

В случае развертывания YARN с агрегированием журналов получить их поможет команда `yarn logs`. Если агрегирование отключено, то к ним все равно можно получить доступ, настроив их хранение в рабочих узлах в течение фиксированного времени с помощью параметра конфигурации `yarn.nodemanager.delete.debug-delay-sec`.

Помимо ручного копирования журналов, в Spark имеется дополнительный «сервер журналов Spark», предоставляющий веб-интерфейс Spark для завершенных заданий.

Подсоединение отладчиков

Хотя файлы журналов, накопители и метрики могут оказать немалую помощь при отладке приложения, иногда совсем не помешает удобный отладочный IDE для Spark. Простейший способ получить его — запустить Spark в локальном режиме и подсоединить отладчик к локальной JVM. К сожалению, не все проблемы реальных кластеров воспроизводимы в локальном режиме, но в подобном случае можно настроить отладку с помощью протокола передачи отладочной информации для Java (Java Debug Wire Protocol, JDWP) (https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/jdwp-spec.html) для отладки в удаленном кластере.

Воспользовавшись свойством `spark.executor.extraJavaOptions` (для исполнителя) или `--driver-java-options` для драйвера, добавьте следующие параметры запуска JVM `-agentlib: jdwp= transport=dt_socket,server=y,address=[debugport]`.

Все остальное после настройки JDWP на работнике или драйвере зависит от конкретного используемого IDE. У IBM имеется руководство по удаленной отладке с помощью eclipse (https://www.ibm.com/developerworks/library/os-eclipse-javadebug/), а применение IntelliJ описано в ответе на сайте Stack Overflow (https://stackoverflow.com/questions/21114066/attach-intellij-idea-debugger-to-a-running-java-process).

Удаленная отладка в языке Python требует модификации кода для запуска отладочной библиотеки. Существует множество вариантов этого: от отладки в интегрированном IDE с применением Jetbrains (https://www.jetbrains.com/help/pycharm/2016.3/remote-debugging.html) и Eclipse (http://www.pydev.org/manual_adv_remote_debugger.html) до более простой удаленной трассировки с помощью rpdb (https://pypi.python.org/pypi/rpdb/) и других инструментов, перечисленных в «Википедии» (https://wiki.python.org/moin/PythonDebuggingTools). Вне зависимости от используемого варианта можно задействовать транслируемую переменную, чтобы гарантировать получение всеми рабочими узлами кода запуска удаленного интерфейса отладки Python.

Добавление -mtrace в путь Python вашего драйвера/работника ничего не даст вследствие жестко «зашитых» в PySpark условий относительно аргументов.

Отладка в блокнотах

В зависимости от блокнота отслеживать журнальную информацию может быть проще или сложнее. При работе в Jupyter с ядром IPython сообщения об ошибках включают только трассу вызовов Java и часто не содержат важную информацию из самого интерпретатора Python. В этом случае необходимо посмотреть в консоль, из которой запускался Jupyter.

Запуск Spark из блокнота самым неожиданным образом может повлиять на использование настроек. Основное отличие заключается в обработке параметров конфигурации драйвера. При работе в среде, расположенной на внешнем сервере (например, в облаке Databricks, IBM Data Science Experience или Spark, находящемся на внешнем сервере Azure компании Microsoft), важно выполнять настройки через механизм, предоставляемый этой средой.

Отладка кода на языке Python

У PySpark имеются дополнительные особенности. Показанная на рис. 7.1 архитектура означает наличие дополнительного уровня сложности. Ярче всего это проявляется в различиях между сообщениями об ошибках в PySpark и сообщениями об аналогичных ошибках в языке Scala (см. примеры П.3 и П.13). Помимо появления этого дополнительного уровня косвенной адресации, конкуренция за ресурсы между различными программами тоже может послужить источником ошибок.

Отложенное вычисление в PySpark даже немного сложнее по сравнению со Scala. Еще больше затуманивает источник ошибки объединение в цепочку вычислений наборов RDD PySpark с целью снизить количество передач данных между Python и JVM и обратно. Переделаем примеры П.1 и П.2, применив язык Python (в результате чего получим примеры П.11 и П.12), и изучим полученные сообщения об ошибках, приведенные в примерах П.13 и П.14.

Пример П.11. Генерация исключения из внутреннего набора RDD (Python)

```
data = sc.parallelize(range(10))
transform1 = data.map(lambda x: x / 0)
transform2 = transform1.map(lambda x: x + 1)
transform2.count()
```

Пример П.12. Генерация исключения из самого верхнего набора RDD (Python)

```
data = sc.parallelize(range(10))
transform1 = data.map(lambda x: x + 1)
transform2 = transform1.map(lambda x: x / 0)
transform2.count()
```

Пример П.13. Сообщение об ошибке во внутреннем наборе RDD (Python)

```
[Stage 0:>                                                          (0 + 0) /
4]17/02/28 22:28:58 ERROR Executor:
Exception in task 3.0 in stage 0.0 (TID 3)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.
    read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1.
    <init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:28:58 ERROR Executor: Exception in task 2.0 in stage 0.0 (TID 2)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
```

```
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:28:58 ERROR Executor: Exception in task 1.0 in stage 0.0 (TID 1)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
```

```
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:28:58 ERROR Executor: Exception in task 0.0 in stage 0.0 (TID 0)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1
    .read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:28:58 WARN TaskSetManager:
```

```
Lost task 0.0 in stage 0.0 (TID 0, localhost, executor driver):
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

17/02/28 22:28:58 ERROR Executor: Exception in task 0.1 in stage 0.0 (TID 7)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
```

```
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:28:58 ERROR TaskSetManager: Task 0 in stage 0.0 failed 2 times;
aborting job
17/02/28 22:28:58 ERROR Executor: Exception in task 2.1 in stage 0.0 (TID 5)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
```

```
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner
    .run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.
java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:28:58 ERROR Executor: Exception in task 3.1 in stage 0.0 (TID 6)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:28:58 ERROR Executor: Exception in task 1.1 in stage 0.0 (TID 4)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
```

```
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "high_performance_pyspark/bad_pyspark.py", line 48, in throwInner
    transform2.count()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in count
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1031, in sum
    return self.mapPartitions(lambda x: [sum(x)]).fold(0, operator.add)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 905, in fold
    vals = self.mapPartitions(func).collect()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 808, in collect
    port = self.ctx._jvm.PythonRDD.collectAndServe(self._jrdd.rdd())
  File "/home/holden/repos/spark/python/lib/py4j-0.10.4-src.zip/py4j/java_gateway.
py",
  line 1133, in __call__
  File "/home/holden/repos/spark/python/pyspark/sql/utils.py", line 63, in deco
    return f(*a, **kw)
  File "/home/holden/repos/spark/python/lib/py4j-0.10.4-src.zip/py4j/protocol.py",
  line 319, in get_return_value
```

```
py4j.protocol.Py4JJavaError:
An error occurred while calling z:org.apache.spark.api.python.PythonRDD
.collectAndServe.
: org.apache.spark.SparkException: Job aborted due to stage failure:
Task 0 in stage 0.0 failed 2 times, most recent failure:
Lost task 0.1 in stage 0.0 (TID 7, localhost, executor driver):
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

Driver stacktrace:
    at org.apache.spark.scheduler
    .DAGScheduler.org$apache$spark$scheduler$DAGScheduler$$
    failJobAndIndependentStages(DAGScheduler.scala:1487)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$abortStage$1.apply(DAGScheduler.scala:1475)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$abortStage$1.apply(DAGScheduler.scala:1474)
    at scala.collection.mutable.
    ResizableArray$class.foreach(ResizableArray.scala:59)
```

```
    at scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:48)
    at org.apache.spark.scheduler.DAGScheduler
    .abortStage(DAGScheduler.scala:1474)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$handleTaskSetFailed$1.apply(DAGScheduler.scala:803)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$handleTaskSetFailed$1.apply(DAGScheduler.scala:803)
    at scala.Option.foreach(Option.scala:257)
    at org.apache.spark.scheduler.DAGScheduler
    .handleTaskSetFailed(DAGScheduler.scala:803)
    at org.apache.spark.scheduler
    .DAGSchedulerEventProcessLoop.doOnReceive(DAGScheduler.scala:1702)
    at org.apache.spark.scheduler
    .DAGSchedulerEventProcessLoop.onReceive(DAGScheduler.scala:1657)
    at org.apache.spark.scheduler
    .DAGSchedulerEventProcessLoop.onReceive(DAGScheduler.scala:1646)
    at org.apache.spark.util.EventLoop$$anon$1.run(EventLoop.scala:48)
    at org.apache.spark.scheduler.DAGScheduler.runJob(DAGScheduler.scala:628)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2011)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2032)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2051)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2076)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1.apply(RDD.scala:936)
    at org.apache.spark.rdd.RDDOperationScope$
    .withScope(RDDOperationScope.scala:151)
    at org.apache.spark.rdd.RDDOperationScope$
    .withScope(RDDOperationScope.scala:112)
    at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
    at org.apache.spark.rdd.RDD.collect(RDD.scala:935)
    at org.apache.spark.api.python.PythonRDD$
    .collectAndServe(PythonRDD.scala:458)
    at org.apache.spark.api.python.PythonRDD
    .collectAndServe(PythonRDD.scala)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl
    .invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl
    .invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at py4j.reflection.MethodInvoker.invoke(MethodInvoker.java:244)
    at py4j.reflection.ReflectionEngine.invoke(ReflectionEngine.java:357)
    at py4j.Gateway.invoke(Gateway.java:280)
    at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.java:132)
    at py4j.commands.CallCommand.execute(CallCommand.java:79)
    at py4j.GatewayConnection.run(GatewayConnection.java:214)
    at java.lang.Thread.run(Thread.java:745)
Caused by: org.apache.spark.api.python.PythonException:
Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
```

```
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1
    .read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    ... 1 more
```

Пример П.14. Сообщение об ошибке во внешнем наборе RDD (Python)

```
17/02/28 22:29:21 ERROR Executor: Exception in task 1.0 in stage 1.0 (TID 9)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
```

```
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1
    .read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:29:21 WARN TaskSetManager:
Lost task 1.0 in stage 1.0 (TID 9, localhost, executor driver):
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
```

```
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

17/02/28 22:29:21 ERROR Executor: Exception in task 0.0 in stage 1.0 (TID 8)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:29:21 ERROR Executor: Exception in task 3.0 in stage 1.0 (TID 11)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
```

```
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:29:21 ERROR Executor: Exception in task 1.1 in stage 1.0 (TID 12)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
```

```
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:29:21 ERROR TaskSetManager:
Task 1 in stage 1.0 failed 2 times; aborting job
17/02/28 22:29:21 ERROR Executor: Exception in task 2.0 in stage 1.0 (TID 10)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1
    .read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:29:21 WARN TaskSetManager:
Lost task 0.1 in stage 1.0 (TID 13, localhost, executor driver):
```

```
TaskKilled (killed intentionally)
17/02/28 22:29:21 WARN TaskSetManager:
Lost task 3.1 in stage 1.0 (TID 14, localhost, executor driver):
TaskKilled (killed intentionally)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "high_performance_pyspark/bad_pyspark.py", line 33, in throwOuter
    transform2.count()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in count
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1031, in sum
    return self.mapPartitions(lambda x: [sum(x)]).fold(0, operator.add)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 905, in fold
    vals = self.mapPartitions(func).collect()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 808, in collect
    port = self.ctx._jvm.PythonRDD.collectAndServe(self._jrdd.rdd())
  File "/home/holden/repos/spark/python/lib/py4j-0.10.4-src.zip/py4j/java_gateway.
py",
  line 1133, in __call__
  File "/home/holden/repos/spark/python/pyspark/sql/utils.py", line 63, in deco
    return f(*a, **kw)
  File "/home/holden/repos/spark/python/lib/py4j-0.10.4-src.zip/py4j/protocol.py",
  line 319, in get_return_value
py4j.protocol.Py4JJavaError:
An error occurred while calling z:org.apache.spark.api.python.PythonRDD
.collectAndServe.
: org.apache.spark.SparkException:
Job aborted due to stage failure: Task 1 in stage 1.0 failed 2 times,
most recent failure: Lost task 1.1 in stage 1.0 (TID 12, localhost,
executor driver): org.apache.spark.api.python.PythonException:
Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1
    .read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
```

```
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

Driver stacktrace:
    at org.apache.spark.scheduler
    .DAGScheduler.org$apache$spark$scheduler$DAGScheduler$$
    failJobAndIndependentStages(DAGScheduler.scala:1487)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$abortStage$1.apply(DAGScheduler.scala:1475)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$abortStage$1.apply(DAGScheduler.scala:1474)
    at scala.collection.mutable.ResizableArray$class
    .foreach(ResizableArray.scala:59)
    at scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:48)
    at org.apache.spark.scheduler
    .DAGScheduler.abortStage(DAGScheduler.scala:1474)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$handleTaskSetFailed$1.apply(DAGScheduler.scala:803)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$handleTaskSetFailed$1.apply(DAGScheduler.scala:803)
    at scala.Option.foreach(Option.scala:257)
    at org.apache.spark.scheduler
    .DAGScheduler.handleTaskSetFailed(DAGScheduler.scala:803)
    at org.apache.spark.scheduler
    .DAGSchedulerEventProcessLoop.doOnReceive(DAGScheduler.scala:1702)
    at org.apache.spark.scheduler
    .DAGSchedulerEventProcessLoop.onReceive(DAGScheduler.scala:1657)
    at org.apache.spark.scheduler
    .DAGSchedulerEventProcessLoop.onReceive(DAGScheduler.scala:1646)
    at org.apache.spark.util.EventLoop$$anon$1.run(EventLoop.scala:48)
    at org.apache.spark.scheduler.DAGScheduler.runJob(DAGScheduler.scala:628)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2011)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2032)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2051)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2076)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1
    .apply(RDD.scala:936)
    at org.apache.spark.rdd.RDDOperationScope$
    .withScope(RDDOperationScope.scala:151)
    at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
    at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
    at org.apache.spark.rdd.RDD.collect(RDD.scala:935)
    at org.apache.spark.api.python.PythonRDD$
```

```
    .collectAndServe(PythonRDD.scala:458)
    at org.apache.spark.api.python.PythonRDD.collectAndServe(PythonRDD.scala)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl
    .invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl
    .invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at py4j.reflection.MethodInvoker.invoke(MethodInvoker.java:244)
    at py4j.reflection.ReflectionEngine.invoke(ReflectionEngine.java:357)
    at py4j.Gateway.invoke(Gateway.java:280)
    at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.java:132)
    at py4j.commands.CallCommand.execute(CallCommand.java:79)
    at py4j.GatewayConnection.run(GatewayConnection.java:214)
    at java.lang.Thread.run(Thread.java:745)
Caused by: org.apache.spark.api.python.PythonException:
Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1
    .read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    ... 1 more
```

Журналирование важно всегда, и, в отличие от приложений Spark для JVM, тут невозможно получить доступ к механизму журналирования `log4j`, чтобы он сделал за нас всю работу[1]. Простейший вариант журналирования для Python — вывод в поток ввода/вывода `stdout`, в результате чего информация об ошибках окажется в потоке ввода/вывода `stderr` журналов работников. Однако данный вариант затрудняет тонкую настройку уровней журналирования. Вместо этого лучше использовать подходящую библиотеку, например стандартную библиотеку `logging`, содержащую функции дописывания в stdout/stderr.

Пользователи YARN могут вместо этого выполнять запись с помощью библиотеки logging. Информация будет записываться в файл, находящийся в каталоге, соответствующем переменной среды LOG_DIRS, которая затем обработается в процессе агрегирования журналов.

Отладку случаев асимметрии наборов RDD в PySpark может осложнить особенность, которой часто не придают большого значения, — *пакетная сериализация*. Обычно ей не уделяют внимания, поскольку при достаточно крупных наборах данных она практически ни на что не влияет. Однако она способна сильно все запутать, если вы последуете общепринятой практике отладки с выборкой небольшого, удобного в обращении поднабора данных и попытаетесь воспроизвести поведение кластера локально.

Использование Python с YARN способно повлечь ошибки перерасхода памяти, которые кажутся ошибками нехватки памяти. В первой части данного приложения мы рассматривали перерасход памяти как потребность программы в определенной дополнительной памяти (см. рис. П.1), но при работе Python весь наш процесс Python должен поместиться внутри этой «перерасходуемой памяти». Данное обстоятельство может осложнить отладку, ведь сообщения об ошибках окажутся одинаковыми для нескольких различных ситуаций.

Первая вероятная причина ошибок памяти — несбалансированные или просто большие секции. При слишком больших размерах секций у работников Python может оказаться недостаточно памяти для загрузки данных. Проверить размер проще всего в веб-интерфейсе. Если секции не сбалансированы, решить проблему часто можно с помощью повторного секционирования, хотя свою роль могут сыграть проблемы асимметрии ключей, обсуждавшиеся в главе 6.

Второй вариант: у нас просто не хватает памяти на перерасход, чтобы покрыть все потребности Python. Выражение «перерасход памяти» в данном случае может сбивать с толку, но в распоряжении у работника Python остается лишь та часть контейнерного пространства, которую не успела израсходовать JVM. Значение по умолчанию

[1] Строго говоря, вы можете получить доступ к нему в драйверной программе с помощью Py4J, но на работниках, где журналирование более важно, шлюз не установлен.

параметра `spark.yarn.executor.memoryOverhead` лишь 384 Мбайт, или 10 % от всего размера контейнера (большее из двух этих значений), что для пользователей Spark обычно не слишком приемлемо. Родственные переменные конфигурации для процесса Application Master (AM) и драйвера (в зависимости от режима развертывания) — `spark.yarn.am.memoryOverhead` и `spark.yarn.driver.memoryOverhead`.

Отладка. Резюме

Хотя отладка в Spark действительно ставит перед нами некие своеобразные и непростые задачи, напомним некоторые из множества ее преимуществ. Одно из главных таково: возможность быстро создать в локальном режиме «фиктивный» кластер для тестирования или отладки, не прибегая к утомительному процессу настройки. Еще одно: наши задания часто работают быстрее, чем традиционные распределенные системы, что позволяет быстро выполнить нужные проверки (или в локальном режиме, или в тестовом кластере) и локализовать источник ошибки. Повторим пожелание из главы 10: пусть же вам придется использовать часть этой книги, посвященную отладке, как можно реже и пусть ваши приключения с Apache Spark приносят только радость.

Холден Карау, Рейчел Уоррен

Эффективный Spark. Масштабирование и оптимизация

Перевел с английского И. Пальти

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*С. Заматевская*
Корректоры	*Е. Павлович, Т. Радецкая*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 191123, Россия, г. Санкт-Петербург,
ул. Радищева, д. 39, к. Д, офис 415. Тел.: +78127037373.
Дата изготовления: 05.2018.
Наименование: книжная продукция.
Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», РБ, 220020, г. Минск, ул. Тимирязева,
д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 26.04.18. Формат 70x100/16. Бумага офсетная. Усл. п. л. 28,380. Тираж 700. Заказ 0000
Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор» .
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87