O'REILLY®

Compliments of talend

FREE CHAPTERS

Advanced Analytics with Spark

PATTERNS FOR LEARNING FROM DATA AT SCALE

Sandy Ryza, Uri Laserson,
Sean Owen & Josh Wills

This Excerpt contains Chapters 1 and 2 of the book *Advanced Analytics with Spark*. The complete book is available at oreilly.com and through other retailers.

Advanced Analytics with Spark

Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills

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

Advanced Analytics with Spark
by Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Marie Beaugureau
Production Editor: Kara Ebrahim
Copyeditor: Kim Cofer
Proofreader: Rachel Monaghan
Indexer: Judy McConville
Interior Designer: David Futato
Cover Designer: Ellie Volckhausen
Illustrator: Rebecca Demarest

April 2015: First Edition

Revision History for the First Edition
2015-03-27: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491912768* for release details.

978-1-491-91276-8

[LSI]

Table of Contents

Preface

Sandy Ryza

I don't like to think I have many regrets, but it's hard to believe anything good came out of a particular lazy moment in 2011 when I was looking into how to best distribute tough discrete optimization problems over clusters of computers. My advisor explained this newfangled Spark thing he had heard of, and I basically wrote off the concept as too good to be true and promptly got back to writing my undergrad thesis in MapReduce. Since then, Spark and I have both matured a bit, but one of us has seen a meteoric rise that's nearly impossible to avoid making "ignite" puns about. Cut to two years later, and it has become crystal clear that Spark is something worth paying attention to.

Spark's long lineage of predecessors, running from MPI to MapReduce, makes it possible to write programs that take advantage of massive resources while abstracting away the nitty-gritty details of distributed systems. As much as data processing needs have motivated the development of these frameworks, in a way the field of big data has become so related to these frameworks that its scope is defined by what these frameworks can handle. Spark's promise is to take this a little further—to make writing distributed programs feel like writing regular programs.

Spark will be great at giving ETL pipelines huge boosts in performance and easing some of the pain that feeds the MapReduce programmer's daily chant of despair ("why? whyyyyy?") to the Hadoop gods. But the exciting thing for me about it has always been what it opens up for complex analytics. With a paradigm that supports iterative algorithms and interactive exploration, Spark is finally an open source framework that allows a data scientist to be productive with large data sets.

I think the best way to teach data science is by example. To that end, my colleagues and I have put together a book of applications, trying to touch on the interactions between the most common algorithms, data sets, and design patterns in large-scale analytics. This book isn't meant to be read cover to cover. Page to a chapter that looks like something you're trying to accomplish, or that simply ignites your interest.

What's in This Book

The first chapter will place Spark within the wider context of data science and big data analytics. After that, each chapter will comprise a self-contained analysis using Spark. The second chapter will introduce the basics of data processing in Spark and Scala through a use case in data cleansing. The next few chapters will delve into the meat and potatoes of machine learning with Spark, applying some of the most common algorithms in canonical applications. The remaining chapters are a bit more of a grab bag and apply Spark in slightly more exotic applications—for example, querying Wikipedia through latent semantic relationships in the text or analyzing genomics data.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/sryza/aas*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Advanced Analytics with Spark* by Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills (O'Reilly). Copyright 2015 Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills, 978-1-491-91276-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like 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, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/advanced-spark*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

It goes without saying that you wouldn't be reading this book if it were not for the existence of Apache Spark and MLlib. We all owe thanks to the team that has built and open sourced it, and the hundreds of contributors who have added to it.

We would like to thank everyone who spent a great deal of time reviewing the content of the book with expert eyes: Michael Bernico, Ian Buss, Jeremy Freeman, Chris Fregly, Debashish Ghosh, Juliet Hougland, Jonathan Keebler, Frank Nothaft, Nick Pentreath, Kostas Sakellis, Marcelo Vanzin, and Juliet Hougland again. Thanks all! We owe you one. This has greatly improved the structure and quality of the result.

I (Sandy) also would like to thank Jordan Pinkus and Richard Wang for helping me with some of the theory behind the risk chapter.

Thanks to Marie Beaugureau and O'Reilly, for the experience and great support in getting this book published and into your hands.

CHAPTER 1
Analyzing Big Data

Sandy Ryza

[Data applications] are like sausages. It is better not to see them being made.
—Otto von Bismarck

- Build a model to detect credit card fraud using thousands of features and billions of transactions.
- Intelligently recommend millions of products to millions of users.
- Estimate financial risk through simulations of portfolios including millions of instruments.
- Easily manipulate data from thousands of human genomes to detect genetic associations with disease.

These are tasks that simply could not be accomplished 5 or 10 years ago. When people say that we live in an age of "big data," they mean that we have tools for collecting, storing, and processing information at a scale previously unheard of. Sitting behind these capabilities is an ecosystem of open source software that can leverage clusters of commodity computers to chug through massive amounts of data. Distributed systems like Apache Hadoop have found their way into the mainstream and have seen widespread deployment at organizations in nearly every field.

But just as a chisel and a block of stone do not make a statue, there is a gap between having access to these tools and all this data, and doing something useful with it. This is where "data science" comes in. As sculpture is the practice of turning tools and raw material into something relevant to nonsculptors, data science is the practice of turning tools and raw data into something that nondata scientists might care about.

Often, "doing something useful" means placing a schema over it and using SQL to answer questions like "of the gazillion users who made it to the third page in our registration process, how many are over 25?" The field of how to structure a data warehouse and organize information to make answering these kinds of questions easy is a rich one, but we will mostly avoid its intricacies in this book.

Sometimes, "doing something useful" takes a little extra. SQL still may be core to the approach, but to work around idiosyncrasies in the data or perform complex analysis, we need a programming paradigm that's a little bit more flexible and a little closer to the ground, and with richer functionality in areas like machine learning and statistics. These are the kinds of analyses we are going to talk about in this book.

For a long time, open source frameworks like R, the PyData stack, and Octave have made rapid analysis and model building viable over small data sets. With fewer than 10 lines of code, we can throw together a machine learning model on half a data set and use it to predict labels on the other half. With a little more effort, we can impute missing data, experiment with a few models to find the best one, or use the results of a model as inputs to fit another. What should an equivalent process look like that can leverage clusters of computers to achieve the same outcomes on huge data sets?

The right approach might be to simply extend these frameworks to run on multiple machines, to retain their programming models and rewrite their guts to play well in distributed settings. However, the challenges of distributed computing require us to rethink many of the basic assumptions that we rely on in single-node systems. For example, because data must be partitioned across many nodes on a cluster, algorithms that have wide data dependencies will suffer from the fact that network transfer rates are orders of magnitude slower than memory accesses. As the number of machines working on a problem increases, the probability of a failure increases. These facts require a programming paradigm that is sensitive to the characteristics of the underlying system: one that discourages poor choices and makes it easy to write code that will execute in a highly parallel manner.

Of course, single-machine tools like PyData and R that have come to recent prominence in the software community are not the only tools used for data analysis. Scientific fields like genomics that deal with large data sets have been leveraging parallel computing frameworks for decades. Most people processing data in these fields today are familiar with a cluster-computing environment called HPC (high-performance computing). Where the difficulties with PyData and R lie in their inability to scale, the difficulties with HPC lie in its relatively low level of abstraction and difficulty of use. For example, to process a large file full of DNA sequencing reads in parallel, we must manually split it up into smaller files and submit a job for each of those files to the cluster scheduler. If some of these fail, the user must detect the failure and take care of manually resubmitting them. If the analysis requires all-to-all operations like sorting the entire data set, the large data set must be streamed through a single node,

or the scientist must resort to lower-level distributed frameworks like MPI, which are difficult to program without extensive knowledge of C and distributed/networked systems. Tools written for HPC environments often fail to decouple the in-memory data models from the lower-level storage models. For example, many tools only know how to read data from a POSIX filesystem in a single stream, making it difficult to make tools naturally parallelize, or to use other storage backends, like databases. Recent systems in the Hadoop ecosystem provide abstractions that allow users to treat a cluster of computers more like a single computer—to automatically split up files and distribute storage over many machines, to automatically divide work into smaller tasks and execute them in a distributed manner, and to automatically recover from failures. The Hadoop ecosystem can automate a lot of the hassle of working with large data sets, and is far cheaper than HPC.

The Challenges of Data Science

A few hard truths come up so often in the practice of data science that evangelizing these truths has become a large role of the data science team at Cloudera. For a system that seeks to enable complex analytics on huge data to be successful, it needs to be informed by, or at least not conflict with, these truths.

First, the vast majority of work that goes into conducting successful analyses lies in preprocessing data. Data is messy, and cleansing, munging, fusing, mushing, and many other verbs are prerequisites to doing anything useful with it. Large data sets in particular, because they are not amenable to direct examination by humans, can require computational methods to even discover what preprocessing steps are required. Even when it comes time to optimize model performance, a typical data pipeline requires spending far more time in feature engineering and selection than in choosing and writing algorithms.

For example, when building a model that attempts to detect fraudulent purchases on a website, the data scientist must choose from a wide variety of potential features: any fields that users are required to fill out, IP location info, login times, and click logs as users navigate the site. Each of these comes with its own challenges in converting to vectors fit for machine learning algorithms. A system needs to support more flexible transformations than turning a 2D array of doubles into a mathematical model.

Second, *iteration* is a fundamental part of the data science. Modeling and analysis typically require multiple passes over the same data. One aspect of this lies *within* machine learning algorithms and statistical procedures. Popular optimization procedures like stochastic gradient descent and expectation maximization involve repeated scans over their inputs to reach convergence. Iteration also matters within the data scientist's own workflow. When data scientists are initially investigating and trying to get a feel for a data set, usually the results of a query inform the next query that should run. When building models, data scientists do not try to get it right in one try.

Choosing the right features, picking the right algorithms, running the right significance tests, and finding the right hyperparameters all require experimentation. A framework that requires reading the same data set from disk each time it is accessed adds delay that can slow down the process of exploration and limit the number of things we get to try.

Third, the task isn't over when a well-performing model has been built. If the point of data science is making data useful to nondata scientists, then a model stored as a list of regression weights in a text file on the data scientist's computer has not really accomplished this goal. Uses of data recommendation engines and real-time fraud detection systems culminate in data applications. In these, models become part of a production service and may need to be rebuilt periodically or even in real time.

For these situations, it is helpful to make a distinction between analytics in the *lab* and analytics in the *factory*. In the lab, data scientists engage in exploratory analytics. They try to understand the nature of the data they are working with. They visualize it and test wild theories. They experiment with different classes of features and auxiliary sources they can use to augment it. They cast a wide net of algorithms in the hopes that one or two will work. In the factory, in building a data application, data scientists engage in operational analytics. They package their models into services that can inform real-world decisions. They track their models' performance over time and obsess about how they can make small tweaks to squeeze out another percentage point of accuracy. They care about SLAs and uptime. Historically, exploratory analytics typically occurs in languages like R, and when it comes time to build production applications, the data pipelines are rewritten entirely in Java or C++.

Of course, everybody could save time if the original modeling code could be actually used in the app for which it is written, but languages like R are slow and lack integration with most planes of the production infrastructure stack, and languages like Java and C++ are just poor tools for exploratory analytics. They lack Read-Evaluate-Print Loop (REPL) environments for playing with data interactively and require large amounts of code to express simple transformations. A framework that makes modeling easy but is also a good fit for production systems is a huge win.

Introducing Apache Spark

Enter Apache Spark, an open source framework that combines an engine for distributing programs across clusters of machines with an elegant model for writing programs atop it. Spark, which originated at the UC Berkeley AMPLab and has since been contributed to the Apache Software Foundation, is arguably the first open source software that makes distributed programming truly accessible to data scientists.

One illuminating way to understand Spark is in terms of its advances over its predecessor, MapReduce. MapReduce revolutionized computation over huge data sets by offering a simple model for writing programs that could execute in parallel across hundreds to thousands of machines. The MapReduce engine achieves near linear scalability—as the data size increases, we can throw more computers at it and see jobs complete in the same amount of time—and is resilient to the fact that failures that occur rarely on a single machine occur all the time on clusters of thousands. It breaks up work into small *tasks* and can gracefully accommodate task failures without compromising the job to which they belong.

Spark maintains MapReduce's linear scalability and fault tolerance, but extends it in three important ways. First, rather than relying on a rigid map-then-reduce format, its engine can execute a more general directed acyclic graph (DAG) of operators. This means that, in situations where MapReduce must write out intermediate results to the distributed filesystem, Spark can pass them directly to the next step in the pipeline. In this way, it is similar to *Dryad*, a descendant of MapReduce that originated at Microsoft Research. Second, it complements this capability with a rich set of transformations that enable users to express computation more naturally. It has a strong developer focus and streamlined API that can represent complex pipelines in a few lines of code.

Third, Spark extends its predecessors with in-memory processing. Its Resilient Distributed Dataset (RDD) abstraction enables developers to materialize any point in a processing pipeline into memory across the cluster, meaning that future steps that want to deal with the same data set need not recompute it or reload it from disk. This capability opens up use cases that distributed processing engines could not previously approach. Spark is well suited for highly iterative algorithms that require multiple passes over a data set, as well as reactive applications that quickly respond to user queries by scanning large in-memory data sets.

Perhaps most importantly, Spark fits well with the aforementioned hard truths of data science, acknowledging that the biggest bottleneck in building data applications is not CPU, disk, or network, but analyst productivity. It perhaps cannot be overstated how much collapsing the full pipeline, from preprocessing to model evaluation, into a single programming environment can speed up development. By packaging an expressive programming model with a set of analytic libraries under a REPL, it avoids the round trips to IDEs required by frameworks like MapReduce and the challenges of subsampling and moving data back and forth from HDFS required by frameworks like R. The more quickly analysts can experiment with their data, the higher likelihood they have of doing something useful with it.

With respect to the pertinence of munging and ETL, Spark strives to be something closer to the Python of big data than the Matlab of big data. As a general-purpose computation engine, its core APIs provide a strong foundation for data transforma-

tion independent of any functionality in statistics, machine learning, or matrix algebra. Its Scala and Python APIs allow programming in expressive general-purpose languages, as well as access to existing libraries.

Spark's in-memory caching makes it ideal for iteration both at the micro and macro level. Machine learning algorithms that make multiple passes over their training set can cache it in memory. When exploring and getting a feel for a data set, data scientists can keep it in memory while they run queries, and easily cache transformed versions of it as well without suffering a trip to disk.

Last, Spark spans the gap between systems designed for exploratory analytics and systems designed for operational analytics. It is often quoted that a data scientist is someone who is better at engineering than most statisticians and better at statistics than most engineers. At the very least, Spark is better at being an operational system than most exploratory systems and better for data exploration than the technologies commonly used in operational systems. It is built for performance and reliability from the ground up. Sitting atop the JVM, it can take advantage of many of the operational and debugging tools built for the Java stack.

Spark boasts strong integration with the variety of tools in the Hadoop ecosystem. It can read and write data in all of the data formats supported by MapReduce, allowing it to interact with the formats commonly used to store data on Hadoop like Avro and Parquet (and good old CSV). It can read from and write to NoSQL databases like HBase and Cassandra. Its stream processing library, Spark Streaming, can ingest data continuously from systems like Flume and Kafka. Its SQL library, SparkSQL, can interact with the Hive Metastore, and a project that is in progress at the time of this writing seeks to enable Spark to be used as an underlying execution engine for Hive, as an alternative to MapReduce. It can run inside YARN, Hadoop's scheduler and resource manager, allowing it to share cluster resources dynamically and to be managed with the same policies as other processing engines like MapReduce and Impala.

Of course, Spark isn't all roses and petunias. While its core engine has progressed in maturity even during the span of this book being written, it is still young compared to MapReduce and hasn't yet surpassed it as the workhorse of batch processing. Its specialized subcomponents for stream processing, SQL, machine learning, and graph processing lie at different stages of maturity and are undergoing large API upgrades. For example, MLlib's pipelines and transformer API model is in progress while this book is being written. Its statistics and modeling functionality comes nowhere near that of single machine languages like R. Its SQL functionality is rich, but still lags far behind that of Hive.

About This Book

The rest of this book is not going to be about Spark's merits and disadvantages. There are a few other things that it will not be either. It will introduce the Spark programming model and Scala basics, but it will not attempt to be a Spark reference or provide a comprehensive guide to all its nooks and crannies. It will not try to be a machine learning, statistics, or linear algebra reference, although many of the chapters will provide some background on these before using them.

Instead, it will try to help the reader get a *feel* for what it's like to use Spark for complex analytics on large data sets. It will cover the entire pipeline: not just building and evaluating models, but cleansing, preprocessing, and exploring data, with attention paid to turning results into production applications. We believe that the best way to teach this is by example, so, after a quick chapter describing Spark and its ecosystem, the rest of the chapters will be self-contained illustrations of what it looks like to use Spark for analyzing data from different domains.

When possible, we will attempt not to just provide a "solution," but to demonstrate the full data science workflow, with all of its iterations, dead ends, and restarts. This book will be useful for getting more comfortable with Scala, more comfortable with Spark, and more comfortable with machine learning and data analysis. However, these are in service of a larger goal, and we hope that most of all, this book will teach you how to approach tasks like those described at the beginning of this chapter. Each chapter, in about 20 measly pages, will try to get as close as possible to demonstrating how to build one of these pieces of data applications.

CHAPTER 2

Introduction to Data Analysis with Scala and Spark

Josh Wills

If you are immune to boredom, there is literally nothing you cannot accomplish.
—David Foster Wallace

Data cleansing is the first step in any data science project, and often the most important. Many clever analyses have been undone because the data analyzed had fundamental quality problems or underlying artifacts that biased the analysis or led the data scientist to see things that weren't really there.

Despite its importance, most textbooks and classes on data science either don't cover data cleansing or only give it a passing mention. The explanation for this is simple: cleansing data is really boring. It is the tedious, dull work that you have to do before you can get to the really cool machine learning algorithm that you've been dying to apply to a new problem. Many new data scientists tend to rush past it to get their data into a minimally acceptable state, only to discover that the data has major quality issues after they apply their (potentially computationally intensive) algorithm and get a nonsense answer as output.

Everyone has heard the saying "garbage in, garbage out." But there is something even more pernicious: getting reasonable-looking answers from a reasonable-looking data set that has major (but not obvious at first glance) quality issues. Drawing significant conclusions based on this kind of mistake is the sort of thing that gets data scientists fired.

One of the most important talents that you can develop as a data scientist is the ability to discover interesting and worthwhile problems in every phase of the data analyt-

ics lifecycle. The more skill and brainpower that you can apply early on in an analysis project, the stronger your confidence will be in your final product.

Of course, it's easy to say all that; it's the data science equivalent of telling children to eat their vegetables. It's much more fun to play with a new tool like Spark that lets us build fancy machine learning algorithms, develop streaming data processing engines, and analyze web-scale graphs. So what better way to introduce you to working with data using Spark and Scala than a data cleansing exercise?

Scala for Data Scientists

Most data scientists have a favorite tool, like R or Python, for performing interactive data munging and analysis. Although they're willing to work in other environments when they have to, data scientists tend to get very attached to their favorite tool, and are always looking to find a way to carry out whatever work they can using it. Introducing them to a new tool that has a new syntax and a new set of patterns to learn can be challenging under the best of circumstances.

There are libraries and wrappers for Spark that allow you to use it from R or Python. The Python wrapper, which is called PySpark, is actually quite good, and we'll cover some examples that involve using it in one of the later chapters in the book. But the vast majority of our examples will be written in Scala, because we think that learning how to work with Spark in the same language in which the underlying framework is written has a number of advantages for you as a data scientist:

It reduces performance overhead.
: Whenever we're running an algorithm in R or Python on top of a JVM-based language like Scala, we have to do some work to pass code and data across the different environments, and oftentimes, things can get lost in translation. When you're writing your data analysis algorithms in Spark with the Scala API, you can be far more confident that your program will run as intended.

It gives you access to the latest and greatest.
: All of Spark's machine learning, stream processing, and graph analytics libraries are written in Scala, and the Python and R bindings can get support for this new functionality much later. If you want to take advantage of all of the features that Spark has to offer (without waiting for a port to other language bindings), you're going to need to learn at least a little bit of Scala, and if you want to be able to extend those functions to solve new problems you encounter, you'll need to learn a little bit more.

It will help you understand the Spark philosophy.
: Even when you're using Spark from Python or R, the APIs reflect the underlying philosophy of computation that Spark inherited from the language in which it was developed—Scala. If you know how to use Spark in Scala, even if you pri-

marily use it from other languages, you'll have a better understanding of the system and will be in a better position to "think in Spark."

There is another advantage to learning how to use Spark from Scala, but it's a bit more difficult to explain because of how different it is from any other data analysis tool. If you've ever analyzed data that you pulled from a database in R or Python, you're used to working with languages like SQL to retrieve the information you want, and then switching into R or Python to manipulate and visualize the data you've retrieved. You're used to using one language (SQL) for retrieving and manipulating lots of data stored in a remote cluster and another language (Python/R) for manipulating and visualizing information stored on your own machine. If you've been doing it for long enough, you probably don't even think about it anymore.

With Spark and Scala, the experience is different, because you're using the same language for *everything*. You're writing Scala to retrieve data from the cluster via Spark. You're writing Scala to manipulate that data locally on your own machine. And then —and this is the really neat part—you can send Scala code into the cluster so that you can perform the exact same transformations that you performed locally on data that is still stored in the cluster. It's difficult to express how transformative it is to do all of your data munging and analysis in a single environment, regardless of where the data itself is stored and processed. It's the sort of thing that you have to experience for yourself to understand, and we wanted to be sure that our examples captured some of that same magic feeling that we felt when we first started using Spark.

The Spark Programming Model

Spark programming starts with a data set or few, usually residing in some form of distributed, persistent storage like the Hadoop Distributed File System (HDFS). Writing a Spark program typically consists of a few related steps:

- Defining a set of transformations on input data sets.
- Invoking actions that output the transformed data sets to persistent storage or return results to the driver's local memory.
- Running local computations that operate on the results computed in a distributed fashion. These can help you decide what transformations and actions to undertake next.

Understanding Spark means understanding the intersection between the two sets of abstractions the framework offers: storage and execution. Spark pairs these abstractions in an elegant way that essentially allows any intermediate step in a data processing pipeline to be cached in memory for later use.

Record Linkage

The problem that we're going to study in this chapter goes by a lot of different names in the literature and in practice: entity resolution, record deduplication, merge-and-purge, and list washing. Ironically, this makes it difficult to find all of the research papers on this topic across the literature in order to get a good overview of solution techniques; we need a data scientist to deduplicate the references to this data cleansing problem! For our purposes in the rest of this chapter, we're going to refer to this problem as *record linkage*.

The general structure of the problem is something like this: we have a large collection of records from one or more source systems, and it is likely that some of the records refer to the same underlying entity, such as a customer, a patient, or the location of a business or an event. Each of the entities has a number of attributes, such as a name, an address, or a birthday, and we will need to use these attributes to find the records that refer to the same entity. Unfortunately, the values of these attributes aren't perfect: values might have different formatting, or typos, or missing information that means that a simple equality test on the values of the attributes will cause us to miss a significant number of duplicate records. For example, let's compare the business listings shown in Table 2-1.

Table 2-1. The challenge of record linkage

Name	Address	City	State	Phone
Josh's Coffee Shop	1234 Sunset Boulevard	West Hollywood	CA	(213)-555-1212
Josh Cofee	1234 Sunset Blvd West	Hollywood	CA	555-1212
Coffee Chain #1234	1400 Sunset Blvd #2	Hollywood	CA	206-555-1212
Coffee Chain Regional Office	1400 Sunset Blvd Suite 2	Hollywood	California	206-555-1212

The first two entries in this table refer to the same small coffee shop, even though a data entry error makes it look as if they are in two different cities (West Hollywood versus Hollywood). The second two entries, on the other hand, are actually referring to different business locations of the same chain of coffee shops that happen to share a common address: one of the entries refers to an actual coffee shop, and the other one refers to a local corporate office location. Both of the entries give the official phone number of corporate headquarters in Seattle.

This example illustrates everything that makes record linkage so difficult: even though both pairs of entries look similar to each other, the criteria that we use to make the duplicate/not-duplicate decision is different for each pair. This is the kind

of distinction that is easy for a human to understand and identify at a glance, but is difficult for a computer to learn.

Getting Started: The Spark Shell and SparkContext

We're going to use a sample data set from the UC Irvine Machine Learning Repository, which is a fantastic source for a variety of interesting (and free) data sets for research and education. The data set we'll be analyzing was curated from a record linkage study that was performed at a German hospital in 2010, and it contains several million pairs of patient records that were matched according to several different criteria, such as the patient's name (first and last), address, and birthday. Each matching field was assigned a numerical score from 0.0 to 1.0 based on how similar the strings were, and the data was then hand-labeled to identify which pairs represented the same person and which did not. The underlying values of the fields themselves that were used to create the data set were removed to protect the privacy of the patients, and numerical identifiers, the match scores for the fields, and the label for each pair (match versus nonmatch) were published for use in record linkage research.

From the shell, let's pull the data from the repository:

```
$ mkdir linkage
$ cd linkage/
$ curl -o donation.zip http://bit.ly/1Aoywaq
$ unzip donation.zip
$ unzip 'block_*.zip'
```

If you have a Hadoop cluster handy, you can create a directory for the block data in HDFS and copy the files from the data set there:

```
$ hadoop fs -mkdir linkage
$ hadoop fs -put block_*.csv linkage
```

The examples and code in this book assume you have Spark 1.2.1 available. Releases can be obtained from the Spark project site. Refer to the Spark documentation for instructions on setting up a Spark environment, whether on a cluster or simply on your local machine.

Now we're ready to launch the `spark-shell`, which is a REPL (read-eval-print loop) for the Scala language that also has some Spark-specific extensions. If you've never seen the term REPL before, you can think of it as something similar to the R environment: it's a place where you can define functions and manipulate data in the Scala programming language.

If you have a Hadoop cluster that runs a version of Hadoop that supports YARN, you can launch the Spark jobs on the cluster by using the value of `yarn-client` for the Spark master:

```
$ spark-shell --master yarn-client
```

However, if you're just running these examples on your personal computer, you can launch a local Spark cluster by specifying `local[N]`, where `N` is the number of threads to run, or `*` to match the number of cores available on your machine. For example, to launch a local cluster that uses eight threads on an eight-core machine:

```
$ spark-shell --master local[*]
```

The examples will work the same way locally. You will simply pass paths to local files, rather than paths on HDFS beginning with `hdfs://`. Note that you will still need to `cp block_*.csv` into your chosen local directory rather than use the directory containing files you unzipped earlier, because it contains a number of other files besides the *.csv* data files.

The rest of the examples in this book will not show a `--master` argument to `spark-shell`, but you will typically need to specify this argument as appropriate for your environment.

You may need to specify additional arguments to make the Spark shell fully utilize your resources. For example, when running Spark with a local master, you can use `--driver-memory 2g` to let the single local process use 2 gigabytes of memory. YARN memory configuration is more complex, and relevant options like `--executor-memory` are explained in the Spark on YARN documentation.

After running one of these commands, you will see a lot of log messages from Spark as it initializes itself, but you should also see a bit of ASCII art, followed by some additional log messages and a prompt:

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.2.1
      /_/
Using Scala version 2.10.4
  (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_67)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
scala>
```

If this is your first time using the Spark shell (or any Scala REPL, for that matter), you should run the `:help` command to list available commands in the shell. `:history` and `:h?` can be helpful for finding the names that you gave to variables or functions that you wrote during a session but can't seem to find at the moment. `:paste` can help you correctly insert code from the clipboard—something you may well want to do while following along with the book and its accompanying source code.

In addition to the note about `:help`, the Spark log messages indicated that "Spark context available as sc." This is a reference to the `SparkContext`, which coordinates the execution of Spark jobs on the cluster. Go ahead and type `sc` at the command line:

```
sc
...
res0: org.apache.spark.SparkContext =
  org.apache.spark.SparkContext@DEADBEEF
```

The REPL will print the string form of the object, and for the `SparkContext` object, this is simply its name plus the hexadecimal address of the object in memory (`DEADBEEF` is a placeholder; the exact value you see here will vary from run to run.)

It's good that the `sc` variable exists, but what exactly do we do with it? `SparkContext` is an object, and as an object, it has methods associated with it. We can see what those methods are in the Scala REPL by typing the name of a variable, followed by a period, followed by tab:

```
sc.[\t]
...
accumulable                accumulableCollection
accumulator                addFile
addJar                     addSparkListener
appName                    asInstanceOf
broadcast                  cancelAllJobs
cancelJobGroup             clearCallSite
clearFiles                 clearJars
clearJobGroup              defaultMinPartitions
defaultMinSplits           defaultParallelism
emptyRDD                   files
getAllPools                getCheckpointDir
getConf                    getExecutorMemoryStatus
getExecutorStorageStatus   getLocalProperty
getPersistentRDDs          getPoolForName
getRDDStorageInfo          getSchedulingMode
hadoopConfiguration        hadoopFile
hadoopRDD                  initLocalProperties
isInstanceOf               isLocal
jars                       makeRDD
master                     newAPIHadoopFile
newAPIHadoopRDD            objectFile
parallelize                runApproximateJob
runJob                     sequenceFile
setCallSite                setCheckpointDir
setJobDescription          setJobGroup
startTime                  stop
submitJob                  tachyonFolderName
textFile                   toString
union                      version
wholeTextFiles
```

The SparkContext has a long list of methods, but the ones that we're going to use most often allow us to create *Resilient Distributed Datasets*, or *RDDs*. An RDD is Spark's fundamental abstraction for representing a collection of objects that can be distributed across multiple machines in a cluster. There are two ways to create an RDD in Spark:

- Using the SparkContext to create an RDD from an external data source, like a file in HDFS, a database table via JDBC, or a local collection of objects that we create in the Spark shell.
- Performing a transformation on one or more existing RDDs, like filtering records, aggregating records by a common key, or joining multiple RDDs together.

RDDs are a convenient way to describe the computations that we want to perform on our data as a sequence of small, independent steps.

Resilient Distributed Datasets

An RDD is laid out across the cluster of machines as a collection of *partitions*, each including a subset of the data. Partitions define the unit of parallelism in Spark. The framework processes the objects within a partition in sequence, and processes multiple partitions in parallel. One of the simplest ways to create an RDD is to use the parallelize method on SparkContext with a local collection of objects:

```
val rdd = sc.parallelize(Array(1, 2, 2, 4), 4)
...
rdd: org.apache.spark.rdd.RDD[Int] = ...
```

The first argument is the collection of objects to parallelize. The second is the number of partitions. When the time comes to compute the objects within a partition, Spark fetches a subset of the collection from the driver process.

To create an RDD from a text file or directory of text files residing in a distributed filesystem like HDFS, we can pass the name of the file or directory to the textFile method:

```
val rdd2 = sc.textFile("hdfs:///some/path.txt")
...
rdd2: org.apache.spark.rdd.RDD[String] = ...
```

When you're running Spark in local mode, the textFile method can access paths that reside on the local filesystem. If Spark is given a directory instead of an individual file, it will consider all of the files in that directory as part of the given RDD. Finally, note that no actual data has been read by Spark or loaded into memory yet, either on our client machine or the cluster. When the time comes to compute the objects within a partition, Spark reads a section (also known as a *split*) of the input

file, and then applies any subsequent transformations (filtering, aggregation, etc.) that we defined via other RDDs.

Our record linkage data is stored in a text file, with one observation on each line. We will use the `textFile` method on `SparkContext` to get a reference to this data as an RDD:

```
val rawblocks = sc.textFile("linkage")
...
rawblocks: org.apache.spark.rdd.RDD[String] = ...
```

There are a few things happening on this line that are worth going over. First, we're declaring a new variable called `rawblocks`. As we can see from the shell, the `rawblocks` variable has a type of `RDD[String]`, even though we never specified that type information in our variable declaration. This is a feature of the Scala programming language called *type inference*, and it saves us a lot of typing when we're working with the language. Whenever possible, Scala figures out what type a variable has based on its context. In this case, Scala looks up the return type from the `textFile` function on the `SparkContext` object, sees that it returns an `RDD[String]`, and assigns that type to the `rawblocks` variable.

Whenever we create a new variable in Scala, we must preface the name of the variable with either `val` or `var`. Variables that are prefaced with `val` are immutable, and cannot be changed to refer to another value once they are assigned, whereas variables that are prefaced with `var` can be changed to refer to different objects of the same type. Watch what happens when we execute the following code:

```
rawblocks = sc.textFile("linkage")
...
<console>: error: reassignment to val

var varblocks = sc.textFile("linkage")
varblocks = sc.textFile("linkage")
```

Attempting to reassign the linkage data to the `rawblocks` `val` threw an error, but reassigning the `varblocks` `var` is fine. Within the Scala REPL, there is an exception to the reassignment of `val`s, because we are allowed to redeclare the same immutable variable, like the following:

```
val rawblocks = sc.textFile("linakge")
val rawblocks = sc.textFile("linkage")
```

In this case, no error is thrown on the second declaration of `rawblocks`. This isn't typically allowed in normal Scala code, but it's fine to do in the shell, and we will make extensive use of this feature throughout the examples in the book.

The REPL and Compilation

In addition to its interactive shell, Spark also supports compiled applications. We typically recommend using *Maven* for compiling and managing dependencies. The GitHub repository included with this book holds a self-contained Maven project setup under the *simplesparkproject/* directory to help you with getting started.

With both the shell and compilation as options, which should you use when testing out and building a data pipeline? It is often useful to start working entirely in the REPL. This enables quick prototyping, faster iteration, and less lag time between ideas and results. However, as the program builds in size, maintaining a monolithic file of code become more onerous, and Scala interpretation eats up more time. This can be exacerbated by the fact that, when you're dealing with massive data, it is not uncommon for an attempted operation to cause a Spark application to crash or otherwise render a `SparkContext` unusable. This means that any work and code typed in so far becomes lost. At this point, it is often useful to take a hybrid approach. Keep the frontier of development in the REPL, and, as pieces of code harden, move them over into a compiled library. You can make the compiled JAR available to `spark-shell` by passing it to the `--jars` property. When done right, the compiled JAR only needs to be rebuilt infrequently, and the REPL allows for fast iteration on code and approaches that still need ironing out.

What about referencing external Java and Scala libraries? To compile code that references external libraries, you need to specify the libraries inside the project's Maven configuration (*pom.xml*). To run code that accesses external libraries, you need to include the JARs for these libraries on the classpath of Spark's processes. A good way to make this happen is to use Maven to package a JAR that includes all of your application's dependencies. You can then reference this JAR when starting the shell by using the `--jars` property. The advantage of this approach is the dependencies only need to be specified once: in the Maven *pom.xml*. Again, the *simplesparkproject/* directory in the GitHub repository shows you how to accomplish this.

SPARK-5341 also tracks development on the capability to specify Maven repositories directly when invoking `spark-shell` and have the JARs from these repositories automatically show up on Spark's classpath.

Bringing Data from the Cluster to the Client

RDDs have a number of methods that allow us to read data from the cluster into the Scala REPL on our client machine. Perhaps the simplest of these is `first`, which returns the first element of the RDD into the client:

```
rawblocks.first
...
res: String = "id_1","id_2","cmp_fname_c1","cmp_fname_c2",...
```

The `first` method can be useful for sanity checking a data set, but we're generally interested in bringing back larger samples of an RDD into the client for analysis. When we know that an RDD only contains a small number of records, we can use the `collect` method to return all of the contents of an RDD to the client as an array. Because we don't know how big the linkage data set is just yet, we'll hold off on doing this right now.

We can strike a balance between `first` and `collect` with the `take` method, which allows us to read a given number of records into an array on the client. Let's use `take` to get the first 10 lines from the linkage data set:

```
val head = rawblocks.take(10)
...
head: Array[String] = Array("id_1","id_2","cmp_fname_c1",...

head.length
...
res: Int = 10
```

Actions

The act of creating an RDD does not cause any distributed computation to take place on the cluster. Rather, RDDs define logical data sets that are intermediate steps in a computation. Distributed computation occurs upon invoking an *action* on an RDD. For example, the `count` action returns the number of objects in an RDD:

```
rdd.count()
14/09/10 17:36:09 INFO SparkContext: Starting job: count ...
14/09/10 17:36:09 INFO SparkContext: Job finished: count ...
res0: Long = 4
```

The `collect` action returns an `Array` with all the objects from the RDD. This `Array` resides in local memory, not on the cluster:

```
rdd.collect()
14/09/29 00:58:09 INFO SparkContext: Starting job: collect ...
14/09/29 00:58:09 INFO SparkContext: Job finished: collect ...
res2: Array[(Int, Int)] = Array((4,1), (1,1), (2,2))
```

Actions need not only return results to the local process. The `saveAsTextFile` action saves the contents of an RDD to persistent storage, such as HDFS:

```
rdd.saveAsTextFile("hdfs:///user/ds/mynumbers")
14/09/29 00:38:47 INFO SparkContext: Starting job:
saveAsTextFile ...
14/09/29 00:38:49 INFO SparkContext: Job finished:
saveAsTextFile ...
```

The action creates a directory and writes out each partition as a file within it. From the command line outside of the Spark shell:

```
hadoop fs -ls /user/ds/mynumbers

-rw-r--r--   3 ds supergroup        0 2014-09-29 00:38 myfile.txt/_SUCCESS
-rw-r--r--   3 ds supergroup        4 2014-09-29 00:38 myfile.txt/part-00000
-rw-r--r--   3 ds supergroup        4 2014-09-29 00:38 myfile.txt/part-00001
```

Remember that `textFile` can accept a directory of text files as input, meaning that a future Spark job could refer to `mynumbers` as an input directory.

The raw form of data that is returned by the Scala REPL can be somewhat hard to read, especially for arrays that contain more than a handful of elements. To make it easier to read the contents of an array, we can use the `foreach` method in conjunction with `println` to print out each value in the array on its own line:

```
head.foreach(println)
...
"id_1","id_2","cmp_fname_c1","cmp_fname_c2","cmp_lname_c1","cmp_lname_c2",
  "cmp_sex","cmp_bd","cmp_bm","cmp_by","cmp_plz","is_match"
37291,53113,0.833333333333333,?,1,?,1,1,1,1,0,TRUE
39086,47614,1,?,1,?,1,1,1,1,1,TRUE
70031,70237,1,?,1,?,1,1,1,1,1,TRUE
84795,97439,1,?,1,?,1,1,1,1,1,TRUE
36950,42116,1,?,1,1,1,1,1,1,1,TRUE
42413,48491,1,?,1,?,1,1,1,1,1,TRUE
25965,64753,1,?,1,?,1,1,1,1,1,TRUE
49451,90407,1,?,1,?,1,1,1,1,0,TRUE
39932,40902,1,?,1,?,1,1,1,1,1,TRUE
```

The `foreach(println)` pattern is one that we will frequently use in this book. It's an example of a common functional programming pattern, where we pass one function (`println`) as an argument to another function (`foreach`) in order to perform some action. This kind of programming style will be familiar to data scientists who have worked with R and are used to processing vectors and lists by avoiding `for` loops and instead using higher-order functions like `apply` and `lapply`. Collections in Scala are similar to lists and vectors in R in that we generally want to avoid `for` loops and instead process the elements of the collection using higher-order functions.

Immediately, we see a couple of issues with the data that we need to address before we begin our analysis. First, the CSV files contain a header row that we'll want to filter out from our subsequent analysis. We can use the presence of the `"id_1"` string in the row as our filter condition, and write a small Scala function that tests for the presence of that string inside of the line:

```
def isHeader(line: String) = line.contains("id_1")
isHeader: (line: String)Boolean
```

Like Python, we declare functions in Scala using the keyword `def`. Unlike Python, we have to specify the types of the arguments to our function; in this case, we have to indicate that the `line` argument is a `String`. The body of the function, which uses the `contains` method for the `String` class to test whether or not the characters `"id_1"` appear anywhere in the string, comes after the equals sign. Even though we had to specify a type for the `line` argument, note that we did not have to specify a return type for the function, because the Scala compiler was able to infer the type based on its knowledge of the `String` class and the fact that the `contains` method returns `true` or `false`.

Sometimes, we will want to specify the return type of a function ourselves, especially for long, complex functions with multiple `return` statements, where the Scala compiler can't necessarily infer the return type itself. We might also want to specify a return type for our function in order to make it easier for someone else reading our code later to be able to understand what the function does without having to reread the entire method. We can declare the return type for the function right after the argument list, like this:

```
def isHeader(line: String): Boolean = {
  line.contains("id_1")
}
isHeader: (line: String)Boolean
```

We can test our new Scala function against the data in the `head` array by using the `filter` method on Scala's `Array` class and then printing the results:

```
head.filter(isHeader).foreach(println)
...
"id_1","id_2","cmp_fname_c1","cmp_fname_c2","cmp_lname_c1",...
```

It looks like our `isHeader` method works correctly; the only result that was returned from applying it to the `head` array via the `filter` method was the header line itself. But of course, what we really want to do is get all of the rows in the data *except* the header rows. There are a few ways that we can do this in Scala. Our first option is to take advantage of the `filterNot` method on the `Array` class:

```
head.filterNot(isHeader).length
...
res: Int = 9
```

We could also use Scala's support for anonymous functions to negate the `isHeader` function from inside `filter`:

```
head.filter(x => !isHeader(x)).length
...
res: Int = 9
```

Anonymous functions in Scala are somewhat like Python's lambda functions. In this case, we defined an anonymous function that takes a single argument called `x` and

passes `x` to the `isHeader` function and returns the negation of the result. Note that we did *not* have to specify any type information for the `x` variable in this instance; the Scala compiler was able to infer that `x` is a `String` from the fact that `head` is an `Array[String]`.

There is nothing that Scala programmers hate more than typing, so Scala has lots of little features that are designed to reduce the amount of typing they have to do. For example, in our anonymous function definition, we had to type the characters `x =>` in order to declare our anonymous function and give its argument a name. For simple anonymous functions like this one, we don't even have to do that; Scala will allow us to use an underscore (`_`) to represent the argument to the anonymous function, so that we can save four characters:

```
head.filter(!isHeader(_)).length
...
res: Int = 9
```

Sometimes, this abbreviated syntax makes the code easier to read because it avoids duplicating obvious identifiers. Sometimes, this shortcut just makes the code cryptic. The code listings use one or the other according to our best judgment.

Shipping Code from the Client to the Cluster

We just saw a wide variety of ways to write and apply functions to data in Scala. All of the code that we executed was done against the data inside the `head` array, which was contained on our client machine. Now we're going to take the code that we just wrote and apply it to the millions of linkage records contained in our cluster and represented by the `rawblocks` RDD in Spark.

Here's what the code looks like to do this; it should feel eerily familiar to you:

```
val noheader = rawblocks.filter(x => !isHeader(x))
```

The syntax that we used to express the filtering computation against the entire data set on the cluster is *exactly the same* as the syntax we used to express the filtering computation against the array of data in `head` on our local machine. We can use the `first` method on the `noheader` RDD to verify that the filtering rule worked correctly:

```
noheader.first
...
res: String = 37291,53113,0.833333333333333,?,1,?,1,1,1,1,0,TRUE
```

This is incredibly powerful. It means that we can interactively develop and debug our data-munging code against a small amount of data that we sample from the cluster, and then ship that code to the cluster to apply it to the entire data set when we're ready to transform the entire data set. Best of all, we never have to leave the shell. There really isn't another tool that gives you this kind of experience.

In the next several sections, we'll use this mix of local development and testing and cluster computation to perform more munging and analysis of the record linkage data, but if you need to take a moment to drink in the new world of awesome that you have just entered, we certainly understand.

Structuring Data with Tuples and Case Classes

Right now, the records in the `head` array and the `noheader` RDD are all strings of comma-separated fields. To make it a bit easier to analyze this data, we'll need to parse these strings into a structured format that converts the different fields into the correct data type, like an integer or double.

If we look at the contents of the `head` array (both the header line and the records themselves), we can see the following structure in the data:

- The first two fields are integer IDs that represent the patients that were matched in the record.
- The next nine values are (possibly missing) double values that represent match scores on different fields of the patient records, such as their names, birthdays, and location.
- The last field is a boolean value (TRUE or FALSE) indicating whether or not the pair of patient records represented by the line was a match.

Like Python, Scala has a built-in *tuple* type that we can use to quickly create pairs, triples, and larger collections of values of different types as a simple way to represent records. For the time being, let's parse the contents of each line into a tuple with four values: the integer ID of the first patient, the integer ID of the second patient, an array of nine doubles representing the match scores (with `NaN` values for any missing fields), and a boolean field that indicates whether or not the fields matched.

Unlike Python, Scala does not have a built-in method for parsing comma-separated strings, so we'll need to do a bit of the legwork ourselves. We can experiment with our parsing code in the Scala REPL. First, let's grab one of the records from the `head` array:

```
val line = head(5)
val pieces = line.split(',')
...
pieces: Array[String] = Array(36950, 42116, 1, ?,...
```

Note that we accessed the elements of the `head` array using parentheses instead of brackets; in Scala, accessing array elements is a function call, not a special operator. Scala allows classes to define a special function named `apply` that is called when we treat an object as if it were a function, so `head(5)` is the same thing as `head.apply(5)`.

We broke up the components of `line` using the `split` function from Java's `String` class, returning an `Array[String]` that we named `pieces`. Now we'll need to convert the individual elements of `pieces` to the appropriate type using Scala's type conversion functions:

```
val id1 = pieces(0).toInt
val id2 = pieces(1).toInt
val matched = pieces(11).toBoolean
```

Converting the `id` variables and the `matched` boolean variable is pretty straightforward once we know about the appropriate `toXYZ` conversion functions. Unlike the `contains` method and `split` method that we worked with earlier, the `toInt` and `toBoolean` methods aren't defined on Java's `String` class. Instead, they are defined in a Scala class called `StringOps` that uses one of Scala's more powerful (and arguably somewhat dangerous) features: *implicit type conversion*. Implicits work like this: if you call a method on a Scala object, and the Scala compiler does not see a definition for that method in the class definition for that object, the compiler will try to convert your object to an instance of a class that *does* have that method defined. In this case, the compiler will see that Java's `String` class does not have a `toInt` method defined, but the `StringOps` class does, and that the `StringOps` class has a method that can convert an instance of the `String` class into an instance of the `StringOps` class. The compiler silently performs the conversion of our `String` object into a `StringOps` object, and then calls the `toInt` method on the new object.

Developers who write libraries in Scala (including the core Spark developers) really like implicit type conversion; it allows them to enhance the functionality of core classes like `String` that are otherwise closed to modification. For a user of these tools, implicit type conversions are more of a mixed bag, because they can make it difficult to figure out exactly where a particular class method is defined. Nonetheless, we're going to encounter implicit conversions throughout our examples, so it's best that we get used to them now.

We still need to convert the double-valued score fields—all nine of them. To convert them all at once, we can use the `slice` method on the Scala `Array` class to extract a contiguous subset of the array, and then use the `map` higher-order function to convert each element of the slice from a `String` to a `Double`:

```
val rawscores = pieces.slice(2, 11)
rawscores.map(s => s.toDouble)
...
java.lang.NumberFormatException: For input string: "?"
  at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:1241)
  at java.lang.Double.parseDouble(Double.java:540)
  ...
```

Oops! We forgot about the "?" entry in the `rawscores` array, and the `toDouble` method in `StringOps` didn't know how to convert it to a `Double`. Let's write a function that will return a `NaN` value whenever it encounters a "?", and then apply it to our `rawscores` array:

```
def toDouble(s: String) = {
  if ("?".equals(s)) Double.NaN else s.toDouble
}
val scores = rawscores.map(toDouble)
scores: Array[Double] = Array(1.0, NaN, 1.0, 1.0, ...
```

There. Much better. Let's bring all of this parsing code together into a single function that returns all of the parsed values in a tuple:

```
def parse(line: String) = {
  val pieces = line.split(',')
  val id1 = pieces(0).toInt
  val id2 = pieces(1).toInt
  val scores = pieces.slice(2, 11).map(toDouble)
  val matched = pieces(11).toBoolean
  (id1, id2, scores, matched)
}
val tup = parse(line)
```

We can retrieve the values of individual fields from our tuple by using the positional functions, starting from `_1`, or via the `productElement` method, which starts counting from `0`. We can also get the size of any tuple via the `productArity` method:

```
tup._1
tup.productElement(0)
tup.productArity
```

Although it is very easy and convenient to create tuples in Scala, addressing all of the elements of a record by position instead of by a meaningful name can make our code difficult to understand. What we would really like is a way of creating a simple record type that would allow us to address our fields by name, instead of by position. Fortunately, Scala provides a convenient syntax for creating these records, called *case classes*. A case class is a simple type of immutable class that comes with implementations of all of the basic Java class methods, like `toString`, `equals`, and `hashCode`, which makes them very easy to use. Let's declare a case class for our record linkage data:

```
case class MatchData(id1: Int, id2: Int,
  scores: Array[Double], matched: Boolean)
```

Now we can update our `parse` method to return an instance of our `MatchData` case class, instead of a tuple:

```
def parse(line: String) = {
  val pieces = line.split(',')
  val id1 = pieces(0).toInt
```

```
  val id2 = pieces(1).toInt
  val scores = pieces.slice(2, 11).map(toDouble)
  val matched = pieces(11).toBoolean
  MatchData(id1, id2, scores, matched)
}
val md = parse(line)
```

There are two things to note here: first, we do not need to specify the keyword `new` in front of `MatchData` when we create a new instance of our case class (another example of how much Scala developers hate typing). Second, our `MatchData` class comes with a built-in `toString` implementation that works great for every field except for the `scores` array.

We can access the fields of the `MatchData` case class by their names now:

```
md.matched
md.id1
```

Now that we have our parsing function tested on a single record, let's apply it to all of the elements in the `head` array, except for the header line:

```
val mds = head.filter(x => !isHeader(x)).map(x => parse(x))
```

Yep, that worked. Now, let's apply our parsing function to the data in the cluster by calling the `map` function on the `noheader` RDD:

```
val parsed = noheader.map(line => parse(line))
```

Remember that unlike the `mds` array that we generated locally, the `parse` function has not actually been applied to the data on the cluster yet. Once we make a call to the `parsed` RDD that requires some output, the `parse` function will be applied to convert each `String` in the `noheader` RDD into an instance of our `MatchData` class. If we make another call to the `parsed` RDD that generates a different output, the `parse` function will be applied to the input data *again*.

This isn't an optimal use of our cluster resources; after the data has been parsed once, we'd like to save the data in its parsed form on the cluster so that we don't have to re-parse it every time we want to ask a new question of the data. Spark supports this use case by allowing us to signal that a given RDD should be cached in memory after it is generated by calling the `cache` method on the instance. Let's do that now for the `parsed` RDD:

```
parsed.cache()
```

Caching

Although the contents of RDDs are transient by default, Spark provides a mechanism for persisting the data in an RDD. After the first time an action requires computing such an RDD's contents, they are stored in memory or disk across the cluster. The next time an action depends on the RDD, it need not be recomputed from its dependencies. Its data is returned from the cached partitions directly:

```
cached.cache()
cached.count()
cached.take(10)
```

The call to `cache` indicates that the RDD should be stored the next time it's computed. The call to `count` computes it initially. The `take` action returns the first 10 elements of the RDD as a local `Array`. When `take` is called, it accesses the cached elements of `cached` instead of recomputing them from their dependencies.

Spark defines a few different mechanisms, or `StorageLevel` values, for persisting RDDs. `rdd.cache()` is shorthand for `rdd.persist(StorageLevel.MEMORY)`, which stores the RDD as unserialized Java objects. When Spark estimates that a partition will not fit in memory, it simply will not store it, and it will be recomputed the next time it's needed. This level makes the most sense when the objects will be referenced frequently and/or require low-latency access, because it avoids any serialization overhead. Its drawback is that it takes up larger amounts of memory than its alternatives. Also, holding on to many small objects puts pressure on Java's garbage collection, which can result in stalls and general slowness.

Spark also exposes a `MEMORY_SER` storage level, which allocates large byte buffers in memory and serializes the RDD contents into them. When we use the right format (more on this in a bit), serialized data usually takes up two to five times less space than its raw equivalent.

Spark can use disk for caching RDDs as well. The `MEMORY_AND_DISK` and `MEMORY_AND_DISK_SER` are similar to the `MEMORY` and `MEMORY_SER` storage levels, respectively. For the latter two, if a partition will not fit in memory, it is simply not stored, meaning that it must be recomputed from its dependencies the next time an action uses it. For the former, Spark spills partitions that will not fit in memory to disk.

Deciding when to cache data can be an art. The decision typically involves trade-offs between space and speed, with the specter of garbage collecting looming overhead to occasionally confound things further. In general, RDDs should be cached when they are likely to be referenced by multiple actions and are expensive to regenerate.

Aggregations

Thus far in the chapter, we've focused on the similar ways that we process data that is on our local machine as well as on the cluster using Scala and Spark. In this section, we'll start to explore some of the differences between the Scala APIs and the Spark ones, especially as they relate to grouping and aggregating data. Most of the differences are about efficiency: when we're aggregating large data sets that are distributed across multiple machines, we're more concerned with transmitting information efficiently than we are when all of the data that we need is available in memory on a single machine.

To illustrate some of the differences, let's start by performing a simple aggregation over our `MatchData` on both our local client and on the cluster with Spark in order to calculate the number of records that are matches versus the number of records that are not. For the local `MatchData` records in the `mds` array, we'll use the `groupBy` method to create a Scala `Map[Boolean, Array[MatchData]]`, where the key is based on the `matched` field in the `MatchData` class:

```
val grouped = mds.groupBy(md => md.matched)
```

Once we have the values in the `grouped` variable, we can get the counts by calling the `mapValues` method on `grouped`, which is like a `map` method that only operates on the values in the `Map` object, and get the `size` of each array:

```
grouped.mapValues(x => x.size).foreach(println)
```

As we can see, all of the entries in our local data are matches, so the only entry returned from the map is the tuple `(true,9)`. Of course, our local data is just a sample of the overall data in the linkage data set; when we apply this grouping to the overall data, we expect to find lots of nonmatches.

When we are performing aggregations on data in the cluster, we always have to be mindful of the fact that the data we are analyzing is stored across multiple machines, and so our aggregations will require moving data over the network that connects the machines. Moving data across the network requires a lot of computational resources: including determining which machines each record will be transferred to, serializing the data, compressing it, sending it over the wire, decompressing and then serializing the results, and finally, performing computations on the aggregated data. To do this quickly, it is important that we try to minimize the amount of data that we move around; the more filtering that we can do to the data before performing an aggregation, the faster we will get an answer to our question.

Creating Histograms

Let's start out by creating a simple histogram to count how many of the `MatchData` records in `parsed` have a value of `true` or `false` for the `matched` field. Fortunately, the `RDD[T]` class defines an action called `countByValue` that performs this kind of computation very efficiently and returns the results to the client as a `Map[T,Long]`. Calling `countByValue` on a projection of the `matched` field from `MatchData` will execute a Spark job and return the results to the client:

```
val matchCounts = parsed.map(md => md.matched).countByValue()
```

Whenever we create a histogram or other grouping of values in the Spark client, especially when the categorical variable in question contains a large number of values, we want to be able to look at the contents of the histogram sorted in different ways, such as by the alphabetical ordering of the keys, or by the numerical counts of the values in ascending or descending order. Although our `matchCounts` `Map` only contains the keys `true` and `false`, let's take a brief look at how to order its contents in different ways.

Scala's `Map` class does not have methods for sorting its contents on the keys or the values, but we can convert a `Map` into a Scala `Seq` type, which does provide support for sorting. Scala's `Seq` is similar to Java's `List` interface, in that it is an iterable collection that has a defined length and the ability to look up values by index:

```
val matchCountsSeq = matchCounts.toSeq
```

Scala Collections

Scala has an extensive library of collections, including lists, sets, maps, and arrays. You can easily convert from one collection type to another using methods like `toList`, `toSet`, and `toArray`.

Our `matchCountsSeq` sequence is made up of elements of type `(String, Long)`, and we can use the `sortBy` method to control which of the indices we use for sorting:

```
matchCountsSeq.sortBy(_._1).foreach(println)
...
(false,5728201)
(true,20931)

matchCountsSeq.sortBy(_._2).foreach(println)
...
(true,20931)
(false,5728201)
```

By default, the `sortBy` function sorts numeric values in ascending order, but it's often more useful to look at the values in a histogram in descending order. We can reverse the sort order of any type by calling the `reverse` method on the sequence before we print it out:

```
matchCountsSeq.sortBy(_._2).reverse.foreach(println)
...
(false,5728201)
(true,20931)
```

When we look at the match counts across the entire data set, we see a significant imbalance between positive and negative matches; less than 0.4% of the input pairs actually match. The implication of this imbalance for our record linkage model is profound: it's likely that any function of the numeric match scores we come up with will have a significant false positive rate (i.e., many pairs of records will look like matches even though they actually are not).

Summary Statistics for Continuous Variables

Spark's `countByValue` action is a great way to create histograms for relatively low cardinality categorical variables in our data. But for continuous variables, like the match scores for each of the fields in the patient records, we'd like to be able to quickly get a basic set of statistics about their distribution, like the mean, standard deviation, and extremal values like the maximum and minimum.

For instances of `RDD[Double]`, the Spark APIs provide an additional set of actions via implicit type conversion, in the same way we saw that the `toInt` method is provided for the `String` class. These implicit actions allow us to extend the functionality of an RDD in useful ways when we have additional information about how to process the values it contains.

Pair RDDs

In addition to the `RDD[Double]` implicit actions, Spark supports implicit type conversion for the `RDD[Tuple2[K, V]]` type that provides methods for performing per-key aggregations like `groupByKey` and `reduceByKey`, as well as methods that enable joining multiple RDDs that have keys of the same type.

One of the implicit actions for `RDD[Double]`, `stats`, will provide us with exactly the summary statistics about the values in the RDD that we want. Let's try it now on the first value in the `scores` array inside of the `MatchData` records in the `parsed` RDD:

```
parsed.map(md => md.scores(0)).stats()
StatCounter = (count: 5749132, mean: NaN, stdev: NaN, max: NaN, min: NaN)
```

Unfortunately, the missing `NaN` values that we are using as placeholders in our arrays are tripping up Spark's summary statistics. Even more unfortunate, Spark does not currently have a nice way of excluding and/or counting up the missing values for us, so we have to filter them out manually using the `isNaN` function from Java's `Double` class:

```
import java.lang.Double.isNaN
parsed.map(md => md.scores(0)).filter(!isNaN(_)).stats()
StatCounter = (count: 5748125, mean: 0.7129, stdev: 0.3887, max: 1.0, min: 0.0)
```

If we were so inclined, we could get all of the statistics for the values in the `scores` array this way, using Scala's `Range` construct to create a loop that would iterate through each index value and compute the statistics for the column, like so:

```
val stats = (0 until 9).map(i => {
  parsed.map(md => md.scores(i)).filter(!isNaN(_)).stats()
})

stats(1)
...
StatCounter = (count: 103698, mean: 0.9000, stdev: 0.2713, max: 1.0, min: 0.0)

stats(8)
...
StatCounter = (count: 5736289, mean: 0.0055, stdev: 0.0741, max: 1.0, min: 0.0)
```

Creating Reusable Code for Computing Summary Statistics

Although this approach gets the job done, it's pretty inefficient; we have to reprocess all of the records in the `parsed` RDD nine times to calculate all of the statistics. As our data sets get larger and larger, the cost of reprocessing all of the data over and over again goes up and up, even when we are caching intermediate results in memory to save on some of the processing time. When we're developing distributed algorithms with Spark, it can really pay off to invest some time in figuring out how we can compute all of the answers we might need in as few passes over the data as possible. In this case, let's figure out a way to write a function that will take in any `RDD[Array[Double]]` we give it and return to us an array that includes both the count of missing values for each index and a `StatCounter` object with the summary statistics of the nonmissing values for each index.

Whenever we expect that some analysis task we need to perform will be useful again and again, it's worth spending some time to develop our code in a way that makes it easy for other analysts to use the solution we come up in their own analyses. To do this, we can write Scala code in a separate file that we can then load into the Spark

shell for testing and validation, and we can then share that file with others once we know that it works.

This is going to require a jump in code complexity. Instead of dealing in individual method calls and functions of a line or two, we need to create proper Scala classes and APIs, and that means using more complex language features.

For our missing value analysis, our first task is to write an analogue of Spark's `StatCounter` class that correctly handles missing values. In a separate shell on your client machine, open a file named *StatsWithMissing.scala*, and copy the following class definitions into the file. We'll walk through the individual fields and methods defined here after the code:

```
import org.apache.spark.util.StatCounter

class NAStatCounter extends Serializable {
  val stats: StatCounter = new StatCounter()
  var missing: Long = 0

  def add(x: Double): NAStatCounter = {
    if (java.lang.Double.isNaN(x)) {
      missing += 1
    } else {
      stats.merge(x)
    }
    this
  }

  def merge(other: NAStatCounter): NAStatCounter = {
    stats.merge(other.stats)
    missing += other.missing
    this
  }

  override def toString = {
    "stats: " + stats.toString + " NaN: " + missing
  }
}

object NAStatCounter extends Serializable {
  def apply(x: Double) = new NAStatCounter().add(x)
}
```

Our `NAStatCounter` class has two member variables: an immutable `StatCounter` instance named `stats`, and a mutable `Long` variable named `missing`. Note that we're marking this class as `Serializable` because we will be using instances of this class inside Spark RDDs, and our job will fail if Spark cannot serialize the data contained inside an RDD.

The first method in the class, `add`, allows us to bring a new `Double` value into the statistics tracked by the `NAStatCounter`, either by recording it as missing if it is `NaN` or adding it to the underlying `StatCounter` if it is not. The `merge` method incorporates the statistics that are tracked by another `NAStatCounter` instance into the current instance. Both of these methods return `this` so that they can be easily chained together.

Finally, we override the `toString` method on our `NAStatCounter` class so that we can easily print out its contents in the Spark shell. Whenever we override a method from a parent class in Scala, we need to prefix the method definition with the `override` keyword. Scala allows a much richer set of method override patterns than Java does, and the `override` keyword helps Scala keep track of which method definition should be used for any given class.

Along with the class definition, we define a *companion object* for `NAStatCounter`. Scala's `object` keyword is used to declare a singleton that can provide helper methods for a class, analogous to the `static` method definitions on a Java class. In this case, the `apply` method provided by the companion object creates a new instance of the `NAStatCounter` class and adds the given `Double` value to the instance before returning it. In Scala, `apply` methods have some special syntactic sugar that allows us to call them without having to type them out explicitly; for example, these two lines do exactly the same thing:

```
val nastats = NAStatCounter.apply(17.29)
val nastats = NAStatCounter(17.29)
```

Now that we have our `NAStatCounter` class defined, let's bring it into the Spark shell by closing and saving the *StatsWithMissing.scala* file and using the `load` command:

```
:load StatsWithMissing.scala
...
Loading StatsWithMissing.scala...
import org.apache.spark.util.StatCounter
defined class NAStatCounter
defined module NAStatCounter
warning: previously defined class NAStatCounter is not a companion to object
NAStatCounter. Companions must be defined together; you may wish to use
:paste mode for this.
```

We get a warning about our companion object not being valid in the incremental compilation mode that the shell uses, but we can verify that a few examples work as we expect:

```
val nas1 = NAStatCounter(10.0)
nas1.add(2.1)
val nas2 = NAStatCounter(Double.NaN)
nas1.merge(nas2)
```

Let's use our new NAStatCounter class to process the scores in the MatchData records within the parsed RDD. Each MatchData instance contains an array of scores of type Array[Double]. For each entry in the array, we would like to have an NAStatCounter instance that tracks how many of the values in that index are NaN along with the regular distribution statistics for the nonmissing values. Given an array of values, we can use the map function to create an array of NAStatCounter objects:

```
val arr = Array(1.0, Double.NaN, 17.29)
val nas = arr.map(d => NAStatCounter(d))
```

Every record in our RDD will have its own Array[Double], which we can translate into an RDD where each record is an Array[NAStatCounter]. Let's go ahead and do that now against the data in the parsed RDD on the cluster:

```
val nasRDD = parsed.map(md => {
  md.scores.map(d => NAStatCounter(d))
})
```

We now need an easy way to aggregate multiple instances of Array[NAStatCounter] into a single Array[NAStatCounter]. We can combine two arrays of the same length using zip. This produces a new Array of the corresponding pairs of elements in the two arrays. Think of a zipper pairing up two corresponding strips of teeth into one fastened strip of interlocked teeth. This can be followed by a map method that uses the merge function on the NAStatCounter class to combine the statistics from both objects into a single instance:

```
val nas1 = Array(1.0, Double.NaN).map(d => NAStatCounter(d))
val nas2 = Array(Double.NaN, 2.0).map(d => NAStatCounter(d))
val merged = nas1.zip(nas2).map(p => p._1.merge(p._2))
```

We can even use Scala's case syntax to break the pair of elements in the zipped array into nicely named variables, instead of using the _1 and _2 methods on the Tuple2 class:

```
val merged = nas1.zip(nas2).map { case (a, b) => a.merge(b) }
```

To perform this same merge operation across all of the records in a Scala collection, we can use the reduce function, which takes an associative function that maps two arguments of type T into a single return value of type T and applies it over and over again to all of the elements in a collection to merge all of the values together. Because the merging logic we wrote earlier is associative, we can apply it with the reduce method to a collection of Array[NAStatCounter] values:

```
val nas = List(nas1, nas2)
val merged = nas.reduce((n1, n2) => {
  n1.zip(n2).map { case (a, b) => a.merge(b) }
})
```

The RDD class also has a reduce action that works the same way as the reduce method we used on the Scala collections, only applied to all of the data that is distributed across the cluster, and the code we use in Spark is identical to the code we just wrote for the List[Array[NAStatCounter]]:

```
val reduced = nasRDD.reduce((n1, n2) => {
  n1.zip(n2).map { case (a, b) => a.merge(b) }
})
reduced.foreach(println)
...
stats: (count: 5748125, mean: 0.7129, stdev: 0.3887,
max: 1.0, min: 0.0) NaN: 1007
stats: (count: 103698, mean: 0.9000, stdev: 0.2713,
max: 1.0, min: 0.0) NaN: 5645434
stats: (count: 5749132, mean: 0.3156, stdev: 0.3342, max: 1.0, min: 0.0) NaN: 0
stats: (count: 2464, mean: 0.3184, stdev: 0.3684,
max: 1.0, min: 0.0) NaN: 5746668
stats: (count: 5749132, mean: 0.9550, stdev: 0.2073, max: 1.0, min: 0.0) NaN: 0
stats: (count: 5748337, mean: 0.2244, stdev: 0.4172, max: 1.0, min: 0.0) NaN: 795
stats: (count: 5748337, mean: 0.4888, stdev: 0.4998, max: 1.0, min: 0.0) NaN: 795
stats: (count: 5748337, mean: 0.2227, stdev: 0.4160, max: 1.0, min: 0.0) NaN: 795
stats: (count: 5736289, mean: 0.0055, stdev: 0.0741,
max: 1.0, min: 0.0) NaN: 12843
```

Let's encapsulate our missing value analysis code into a function in the *StatsWithMissing.scala* file that allows us to compute these statistics for any RDD[Array[Double]] by editing the file to include this block of code:

```
import org.apache.spark.rdd.RDD

def statsWithMissing(rdd: RDD[Array[Double]]): Array[NAStatCounter] = {
  val nastats = rdd.mapPartitions((iter: Iterator[Array[Double]]) => {
    val nas: Array[NAStatCounter] = iter.next().map(d => NAStatCounter(d))
    iter.foreach(arr => {
      nas.zip(arr).foreach { case (n, d) => n.add(d) }
    })
    Iterator(nas)
  })
  nastats.reduce((n1, n2) => {
    n1.zip(n2).map { case (a, b) => a.merge(b) }
  })
}
```

Note that instead of calling the map function to generate an Array[NAStatCounter] for each record in the input RDD, we're calling the slightly more advanced mapPartitions function, which allows us to process *all* of the records within a partition of the input RDD[Array[Double]] via an Iterator[Array[Double]]. This allows us to create a single instance of Array[NAStatCounter] for each partition of the data and then update its state using the Array[Double] values that are returned by the given iterator, which is a more efficient implementation. Indeed, our statsWithMissing method

is now very similar to how the Spark developers implemented the `stats` method for instances of type `RDD[Double]`.

Simple Variable Selection and Scoring

With the `statsWithMissing` function, we can analyze the differences in the distribution of the arrays of scores for both the matches and the nonmatches in the `parsed` RDD:

```
val statsm = statsWithMissing(parsed.filter(_.matched).map(_.scores))
val statsn = statsWithMissing(parsed.filter(!_.matched).map(_.scores))
```

Both the `statsm` and `statsn` arrays have identical structure, but they describe different subsets of our data: `statsm` contains the summary statistics on the `scores` array for matches, while `statsn` does the same thing for nonmatches. We can use the differences in the values of the columns for matches and nonmatches as a simple bit of analysis to help us come up with a scoring function for discriminating matches from nonmatches purely in terms of these match scores:

```
statsm.zip(statsn).map { case(m, n) =>
  (m.missing + n.missing, m.stats.mean - n.stats.mean)
}.foreach(println)
...
((1007, 0.2854...), 0)
((5645434,0.09104268062279874), 1)
((0,0.6838772482597568), 2)
((5746668,0.8064147192926266), 3)
((0,0.03240818525033484), 4)
((795,0.7754423117834044), 5)
((795,0.5109496938298719), 6)
((795,0.7762059675300523), 7)
((12843,0.9563812499852178), 8)
```

A good feature has two properties: it tends to have significantly different values for matches and nonmatches (so the difference between the means will be large) and it occurs often enough in the data that we can rely on it to be regularly available for any pair of records. By this measure, Feature 1 isn't very useful: it's missing a lot of the time, and the difference in the mean value for matches and nonmatches is relatively small—0.09, for a score that ranges from 0 to 1. Feature 4 also isn't particularly helpful. Even though it's available for any pair of records, the difference in means is just 0.03.

Features 5 and 7, on the other hand, are excellent: they almost always occur for any pair of records, and there is a very large difference in the mean values (over 0.77 for both features.) Features 2, 6, and 8 also seem beneficial: they are generally available in the data set and the difference in mean values for matches and nonmatches are substantial.

Features 0 and 3 are more of a mixed bag: Feature 0 doesn't discriminate all that well (the difference in the means is only 0.28), even though it's usually available for a pair of records, while Feature 3 has a large difference in the means, but it's almost always missing. It's not quite obvious under what circumstances we should include these features in our model based on this data.

For now, we're going to use a simple scoring model that ranks the similarity of pairs of records based on the sums of the values of the obviously good features: 2, 5, 6, 7, and 8. For the few records where the values of these features are missing, we'll use 0 in place of the `NaN` value in our sum. We can get a rough feel for the performance of our simple model by creating an RDD of scores and match values and evaluating how well the score discriminates between matches and nonmatches at various thresholds:

```
def naz(d: Double) = if (Double.NaN.equals(d)) 0.0 else d
case class Scored(md: MatchData, score: Double)
val ct = parsed.map(md => {
  val score = Array(2, 5, 6, 7, 8).map(i => naz(md.scores(i))).sum
  Scored(md, score)
})
```

Using a high threshold value of 4.0, meaning that the average of the five features was 0.8, we filter out almost all of the nonmatches while keeping over 90% of the matches:

```
ct.filter(s => s.score >= 4.0).map(s => s.md.matched).countByValue()
...
Map(false -> 637, true -> 20871)
```

Using the lower threshold of 2.0, we can ensure that we capture *all* of the known matching records, but at a substantial cost in terms of false positives:

```
ct.filter(s => s.score >= 2.0).map(s => s.md.matched).countByValue()
...
Map(false -> 596414, true -> 20931)
```

Even though the number of false positives is higher than we would like, this more generous filter still removes 90% of the nonmatching records from our consideration while including every positive match. Even though this is pretty good, it's possible to do even better; see if you can find a way to use some of the other values from the `scores` array (both missing and not) to come up with a scoring function that successfully identifies every `true` match at the cost of less than 100 false positives.

Where to Go from Here

If this chapter was your first time carrying out data preparation and analysis with Scala and Spark, we hope that you got a feel for what a powerful foundation these tools provide. If you have been using Scala and Spark for a while, we hope that you will pass this chapter along to your friends and colleagues as a way of introducing them to that power as well.

Our goal for this chapter was to provide you with enough Scala knowledge to be able to understand and carry out the rest of the examples in this book. If you are the kind of person who learns best through practical examples, your next step is to continue on to the next set of chapters, where we will introduce you to MLlib, the machine learning library designed for Spark.

As you become a seasoned user of Spark and Scala for data analysis, it's likely that you will reach a point where you begin to build tools and libraries that are designed to help other analysts and data scientists apply Spark to solve their own problems. At that point in your development, it would be helpful to pick up additional books on Scala, like *Programming Scala* by Dean Wampler and Alex Payne, and *The Scala Cookbook* by Alvin Alexander (both from O'Reilly).

About the Authors

Sandy Ryza is a Senior Data Scientist at Cloudera and active contributor to the Apache Spark project. He recently led Spark development at Cloudera and now spends his time helping customers with a variety of analytic use cases on Spark. He is also a member of the Hadoop Project Management Committee.

Uri Laserson is a Senior Data Scientist at Cloudera, where he focuses on Python in the Hadoop ecosystem. He also helps customers deploy Hadoop on a wide range of problems, focusing on life sciences and health care. Previously, Uri cofounded Good Start Genetics, a next-generation diagnostics company while working toward a PhD in biomedical engineering at MIT.

Sean Owen is Director of Data Science for EMEA at Cloudera. He has been a committer and significant contributor to the Apache Mahout machine learning project, and authored its "Taste" recommender framework. Sean is an Apache Spark committer. He created the Oryx (formerly Myrrix) project for real-time large-scale learning on Hadoop, built on Spark, Spark Streaming, and Kafka.

Josh Wills is Senior Director of Data Science at Cloudera, working with customers and engineers to develop Hadoop-based solutions across a wide range of industries. He is the founder and VP of the Apache Crunch project for creating optimized MapReduce and Spark pipelines in Java. Prior to joining Cloudera, Josh worked at Google, where he worked on the ad auction system and then led the development of the analytics infrastructure used in Google+.

Colophon

The animal on the cover of *Advanced Analytics with Spark* is a peregrine falcon (*Falco peregrinus*); these falcons are among the world's most common birds of prey and live on all continents except Antarctica. They can survive in a wide variety of habitats including urban cities, the tropics, deserts, and the tundra. Some migrate long distances from their wintering areas to their summer nesting areas.

Peregrine falcons are the fastest-flying birds in the world—they are able to dive at 200 miles per hour. They eat other birds such as songbirds and ducks, as well as bats, and they catch their prey in mid-air.

Adults have blue-gray wings, dark brown backs, a buff colored underside with brown spots, and white faces with a black tear stripe on their cheeks. They have a hooked beak and strong talons. Their name comes from the Latin word *peregrinus*, which means "to wander." Peregrines are favored by falconers, and have been used in that sport for many centuries.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from Lydekker's *Royal Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

WP199-EN