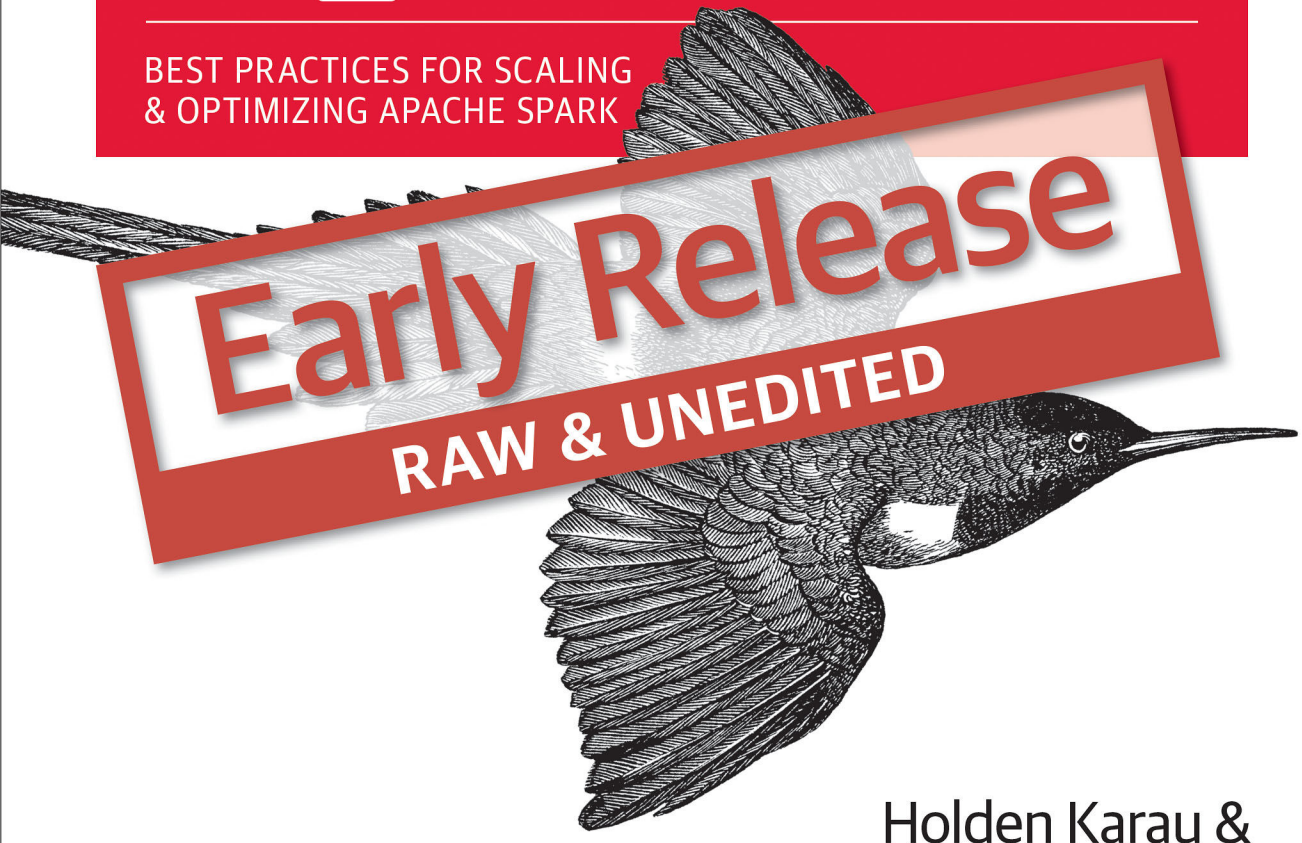
O'REILLY®

High Performance Spark

BEST PRACTICES FOR SCALING & OPTIMIZING APACHE SPARK



Early Release

RAW & UNEDITED

Holden Karau &
Rachel Warren

FIRST EDITION

High Performance Spark

Holden Karau and Rachel Warren

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

High Performance Spark
by Holden Karau and Rachel Warren

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: Shannon Cutt
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

July 2016: First Edition

Revision History for the First Edition
2016-03-21: First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491943205* for release details.

978-1-491-94320-5

[FILL IN]

Table of Contents

Preface

Who Is This Book For?

This book is for data engineers and data scientists who are looking to get the most out of Spark. If you've been working with Spark and invested in Spark but your experience so far has been mired by memory errors and mysterious, intermittent failures, this book is for you. If you have been using Spark for some exploratory work or experimenting with it on the side but haven't felt confident enough to put it into production, this book may help. If you are enthusiastic about Spark but haven't seen the performance improvements from it that you expected, we hope this book can help. This book is intended for those who have some working knowledge of Spark and may be difficult to understand for those with little or no experience with Spark or distributed computing. For recommendations of more introductory literature see "Supporting Books & Materials" on page vi.

We expect this text will be most useful to those who care about optimizing repeated queries in production, rather than to those who are doing merely exploratory work. While writing highly performant queries is perhaps more important to the data engineer, writing those queries with Spark, in contrast to other frameworks, requires a good knowledge of the data, usually more intuitive to the data scientist. Thus it may be more useful to a data engineer who may be less experienced with thinking critically about the statistical nature, distribution, and layout of your data when considering performance. We hope that this book will help data engineers think more critically about their data as they put pipelines into production. We want to help our readers ask questions such as: "How is my data distributed?" "Is it skewed?", "What is the range of values in a column?", "How do we expect a given value to group?" "Is it skewed?". And to apply the answers to those questions to the logic of their Spark queries.

However, even for data scientists using Spark mostly for exploratory purposes, this book should cultivate some important intuition about writing performant Spark queries, so that as the scale of the exploratory analysis inevitably grows, you may have

a better shot of getting something to run the first time. We hope to guide data scientists, even those who are already comfortable thinking about data in a distributed way, to think critically about how their programs are evaluated, empowering them to explore their data more fully. more quickly, and to communicate effectively with anyone helping them put their algorithms into production.

Regardless of your job title, it is likely that the amount of data with which you are working is growing quickly. Your original solutions may need to be scaled, and your old techniques for solving new problems may need to be updated. We hope this book will help you leverage Apache Spark to tackle new problems more easily and old problems more efficiently.

Early Release Note

You are reading an early release version of High Performance Spark, and for that, we thank you! If you find errors, mistakes, or have ideas for ways to improve this book, please reach out to us at *high-performance-spark@googlegroups.com*. If you wish to be included in a "thanks" section in future editions of the book, please include your preferred display name.

This is an early release. While there are always mistakes and omissions in technical books, this is especially true for an early release book.

Supporting Books & Materials

For data scientists and developers new to Spark, *Learning Spark* by Karau, Konwinski, Wendel, and Zaharia is an excellent introduction, [1] and "Advanced Analytics with Spark" by Sandy Ryza, Uri Laserson, Sean Owen, Josh Wills is a great book for interested data scientists.

Beyond books, there is also a collection of intro-level Spark training material available. For individuals who prefer video, Paco Nathan has an excellent introduction video series on O'Reilly. Commercially, Databricks as well as Cloudera and other Hadoop/Spark vendors offer Spark training. Previous recordings of Spark camps, as well as many other great resources, have been posted on the Apache Spark documentation page.

1 albeit we may be biased

If you don't have experience with Scala, we do our best to convince you to pick up Scala in Chapter 1, and if you are interested in learning, "Programming Scala, 2nd Edition" by Dean Wampler, Alex Payne is a good introduction.[2]

Conventions Used in this Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

2 Although it's important to note that some of the practices suggested in this book are not common practice in Spark code.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download from the High Performance Spark GitHub Repository and some of the testing code is available at the "Spark Testing Base" Github Repository. and the Spark Validator Repo.

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. The code is also available under an Apache 2 License. Incorporating a significant amount of example code from this book into your product's documentation may require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

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 the Authors

For feedback on the early release, e-mail us at *high-performance-spark@googlegroups.com*. For random ramblings, occasionally about Spark, follow us on twitter:

Holden: *http://twitter.com/holdenkarau*

Rachel: *https://twitter.com/warre_n_peace*

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://www.oreilly.com/catalog/<catalog page>*.

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

The authors would like to acknowledge everyone who has helped with comments and suggestions on early drafts of our work. Special thanks to Anya Bida and Jakob Odersky for reviewing early drafts and diagrams. We'd also like to thank Mahmoud Hanafy for reviewing and improving the sample code as well as early drafts. We'd also like to thank Michael Armbrust for reviewing and providing feedback on early drafts of the SQL chapter.

We'd also like to thank our respective employers for being understanding as we've worked on this book. Especially Lawrence Spracklen who insisted we mention him here :p.

CHAPTER 1

Introduction to High Performance Spark

This chapter provides an overview of what we hope you will be able to learn from this book and does its best to convince you to learn Scala. Feel free to skip ahead to Chapter 2 if you already know what you're looking for and use Scala (or have your heart set on another language).

Spark Versions

Spark follows semantic versioning with the standard [MAJOR].[MINOR].[MAINTENANCE] with API stability for public non-experimental non-developer APIs. Many of these experimental components are some of the more exciting from a performance standpoint, including Datasets-- Spark SQL's new structured, strongly-typed, data abstraction. Spark also tries for binary API compatibility between releases, using MiMa; so if you are using the stable API you generally should not need to recompile to run our job against a new version of Spark.

This book is created using the Spark 1.6 APIs (and the final version will be updated to 2.0) - but much of the code will work in earlier versions of Spark as well. In places where this is not the case we have attempted to call that out.

What is Spark and Why Performance Matters

Apache Spark is a high-performance, general-purpose distributed computing system that has become the most active Apache open-source project, with more than 800

active contributors. [1] Spark enables us to process large quantities of data, beyond what can fit on a single machine, with a high-level, relatively easy-to-use API. Spark's design and interface are unique, and it is one of the fastest systems of its kind. Uniquely, Spark allows us to write the logic of data transformations and machine learning algorithms in a way that is parallelizable, but relatively system agnostic. So it is often possible to write computations which are fast for distributed storage systems of varying kind and size.

However, despite its many advantages and the excitement around Spark, the simplest implementation of many common data science routines in Spark can be much slower and much less robust than the best version. Since the computations we are concerned with may involve data at a very large scale, the time and resources that gains from tuning code for performance are enormous. Performance that does not just mean run faster; often at this scale it means getting something to run at all. It is possible to construct a Spark query that fails on gigabytes of data but, when refactored and adjusted with an eye towards the structure of the data and the requirements of the cluster succeeds on the same system with terabytes of data. In the author's experience, writing production Spark code, we have seen the same tasks, run on the same clusters, run 100x faster using some of the optimizations discussed in this book. In terms of data processing, time is money, and we hope this book pays for itself through a reduction in data infrastructure costs and developer hours.

Not all of these techniques are applicable to every use case. Especially because Spark is highly configurable, but also exposed at a higher level than other computational frameworks of comparable power, we can reap tremendous benefits just by becoming more attuned to the shape and structure of your data. Some techniques can work well on certain data sizes or even certain key distributions but not all. The simplest example of this can be how for many problems, using `groupByKey` in Spark can very easily cause the dreaded out of memory exceptions, but for data with few duplicates this operation can be almost the same. Learning to understand your particular use case and system and how Spark will interact with it is a must to solve the most complex data science problems with Spark.

What You Can Expect to Get from This Book

Our hope is that this book will help you take your Spark queries and make them faster, able to handle larger data sizes, and use fewer resources. This book covers a broad range of tools and scenarios. You will likely pick up some techniques which might not apply to the problems you are working with, but which might apply to a problem in the future and which may help shape your understanding of Spark more

1 From *http://spark.apache.org/* "Since 2009, more than 800 developers have contributed to Spark".

generally. The chapters in this book are written with enough context to allow the book to be used as a reference; however, the structure of this book is intentional and reading the sections in order should give you not only a few scattered tips but a comprehensive understanding of Apache Spark and how to make it sing.

It's equally important to point out what you will likely not get from this book. This book is not intended to be an introduction to Spark or Scala; several other books and video series are available to get you started. The authors may be a little biased in this regard, but we think "Learning Spark" by Karau, Konwinski, Wendel, and Zaharia as well as Paco Nathan's Introduction to Apache Spark video series are excellent options for Spark beginners. While this book is focused on performance, it is not an operations book, so topics like setting up a cluster and multi-tenancy are not covered. We are assuming that you already have a way to use Spark in your system and won't provide much assistance in making higher-level architecture decisions. There are future books in the works, by other authors, on the topic of Spark operations that may be done by the time you are reading this one. If operations are your show, or if there isn't anyone responsible for operations in your organization, we hope those books can help you. ==== Why Scala?

In this book, we will focus on Spark's Scala API and assume a working knowledge of Scala. Part of this decision is simply in the interest of time and space; we trust readers wanting to use Spark in another language will be able to translate the concepts used in this book without presenting the examples in Java and Python. More importantly, it is the belief of the authors that "serious" performant Spark development is most easily achieved in Scala. To be clear these reasons are very specific to using Spark with Scala; there are many more general arguments for (and against) Scala's applications in other contexts.

To Be a Spark Expert You Have to Learn a Little Scala Anyway

Although Python and Java are more commonly used languages, learning Scala is a worthwhile investment for anyone interested in delving deep into Spark development. Spark's documentation can be uneven. However, the readability of the codebase is world-class. Perhaps more than with other frameworks, the advantages of cultivating a sophisticated understanding of the Spark code base is integral to the advanced Spark user. Because Spark is written in Scala, it will be difficult to interact with the Spark source code without the ability, at least, to read Scala code. Furthermore, the methods in the RDD class closely mimic those in the Scala collections API. RDD functions, such as `map`, `filter`, `flatMap`, `reduce`, and `fold`, have nearly identical specifications to their Scala equivalents [2] Fundamentally Spark is a functional framework,

2 Although, as we explore in this book, the performance implications and evaluation semantics are quite different.

relying heavily on concepts like immutability and lambda definition, so using the Spark API may be more intuitive with some knowledge of the functional programming.

The Spark Scala API is Easier to Use Than the Java API

Once you have learned Scala, you will quickly find that writing Spark in Scala is less painful than writing Spark in Java. First, writing Spark in Scala is significantly more concise than writing Spark in Java since Spark relies heavily on in line function definitions and lambda expressions, which are much more naturally supported in Scala (especially before Java 8). Second, the Spark shell can be a powerful tool for debugging and development, and it is obviously not available in a compiled language like Java.

Scala is More Performant Than Python

It can be attractive to write Spark in Python, since it is easy to learn, quick to write, interpreted, and includes a very rich set of data science tool kits. However, Spark code written in Python is often slower than equivalent code written in the JVM, since Scala is statically typed, and the cost of JVM communication (from Python to Scala) can be very high. Last, Spark features are generally written in Scala first and then translated into Python, so to use cutting edge Spark functionality, you will need to be in the JVM; Python support for MLlib and Spark Streaming are particularly behind.

Why Not Scala?

There are several good reasons, to develop with Spark in other languages. One of the more important constant reason is developer/team preference. Existing code, both internal and in libraries, can also be a strong reason to use a different language. Python is one of the most supported languages today. While writing Java code can be clunky and sometimes lag slightly in terms of API, there is very little performance cost to writing in another JVM language (at most some object conversions). [3]

While all of the examples in this book are presented in Scala for the final release, we will port many of the examples from Scala to Java and Python where the differences in implementation could be important. These will be available (over time) at our Github. If you find yourself wanting a specific example ported please either e-mail us or create an issue on the github repo.

3 Of course, in performance, every rule has its exception. `mapPartitions` in Spark 1.6 and earlier in Java suffers some sever performance restrictions we discuss in ???.

Spark SQL does much to minimize performance difference when using a non-JVM language. ??? looks at options to work effectively in Spark with languages outside of the JVM, including Spark's supported languages of Python and R. This section also offers guidance on how to use Fortran, C, and GPU specific code to reap additional performance improvements. Even if we are developing most of our Spark application in Scala, we shouldn't feel tied to doing everything in Scala, because specialized libraries in other languages can be well worth the overhead of going outside the JVM.

Learning Scala

If after all of this we've convinced you to use Scala, there are several excellent options for learning Scala. The current version of Spark is written against Scala 2.10 and cross-compiled for 2.11 (with the future changing to being written for 2.11 and cross-compiled against 2.10). Depending on how much we've convinced you to learn Scala, and what your resources are, there are a number of different options ranging from books to MOOCs to professional training.

For books, *Programming Scala, 2nd Edition* by Dean Wampler and Alex Payne can be great, although much of the actor system references are not relevant while working in Spark. The Scala language website also maintains a list of Scala books.

In addition to books focused on Spark, there are online courses for learning Scala. *Functional Programming Principles in Scala*, taught by Martin Ordersky, its creator, is on Coursera as well as Introduction to Functional Programming on edX. A number of different companies also offer video-based Scala courses, none of which the authors have personally experienced or recommend.

For those who prefer a more interactive approach, professional training is offered by a number of different companies including, Typesafe. While we have not directly experienced Typesafe training, it receives positive reviews and is known especially to help bring a team or group of individuals up to speed with Scala for the purposes of working with Spark.

Conclusion

Although you will likely be able to get the most out of Spark performance if you have an understanding of Scala, working in Spark does not require a knowledge of Scala. For those whose problems are better suited to other languages or tools, techniques for working with other languages will be covered in ???. This book is aimed at individuals who already have a grasp of the basics of Spark, and we thank you for choosing *High Performance Spark* to deepen your knowledge of Spark. The next chapter will introduce some of Spark's general design and evaluation paradigm which is important to understanding how to efficiently utilize Spark.

CHAPTER 2
How Spark Works

This chapter introduces Spark's place in the big data ecosystem and its overall design. Spark is often considered an alternative to Apache MapReduce, since Spark can also be used for distributed data processing with Hadoop. [1], packaged with the distributed file system Apache Hadoop.] As we will discuss in this chapter, Spark's design principals are quite different from MapReduce's and Spark doe not need to be run in tandem with Apache Hadoop. Furthermore, while Spark has inherited parts of its API, design, and supported formats from existing systems, particularly DraydLINQ, Spark's internals, especially how it handles failures, differ from many traditional systems. [2] Spark's ability to leverage lazy evaluation within memory computations make it particularly unique. Spark's creators believe it to be the first high-level programing language for fast, distributed data processing. [3] Understanding the general design principals behind Spark will be useful for understanding the performance of Spark jobs.

To get the most out of Spark, it is important to understand some of the principles used to design Spark and, at a cursory level, how Spark programs are executed. In this chapter, we will provide a broad overview of Spark's model of parallel computing and

1 MapReduce is a programmatic paradigm that defines programs in terms of *map* procedures that filter and sort data onto the nodes of a distributed system, and *reduce* procedures that aggregate the data on the mapper nodes. Implementations of MapReduce have been written in many languages, but the term usually refers to a popular implementation called link::http://hadoop.apache.org/[*Hadoop MapReduce*

2 DryadLINQ is a Microsoft research project which puts the .NET Language Integrated Query (LINQ) on top of the Dryad distributed execution engine. Like Spark, The DraydLINQ API defines an object representing a distributed dataset and exposes functions to transform data as methods defined on the dataset object. DraydLINQ is lazily evaluated and its scheduler is similar to Spark's however, it doesn't use in memory storage. For more information see the DraydLINQ documentation.

3 See the original Spark Paper.

a thorough explanation of the Spark scheduler and execution engine. We will refer to the concepts in this chapter throughout the text. Further, this explanation will help you get a more precise understanding of some of the terms you've heard tossed around by other Spark users and in the Spark documentation.

How Spark Fits into the Big Data Ecosystem

Apache Spark is an open source framework that provides highly generalizable methods to process data in parallel. On its own, Spark is not a data storage solution. Spark can be run locally, on a single machine with a single JVM (called local mode). More often Spark is used in tandem with a distributed storage system to write the data processed with Spark (such as HDFS, Cassandra, or S3) and a cluster manager to manage the distribution of the application across the cluster. Spark currently supports three kinds of cluster managers: the manager included in Spark, called the Standalone Cluster Manager, which requires Spark to be installed in each node of a cluster, Apache Mesos; and Hadoop YARN.

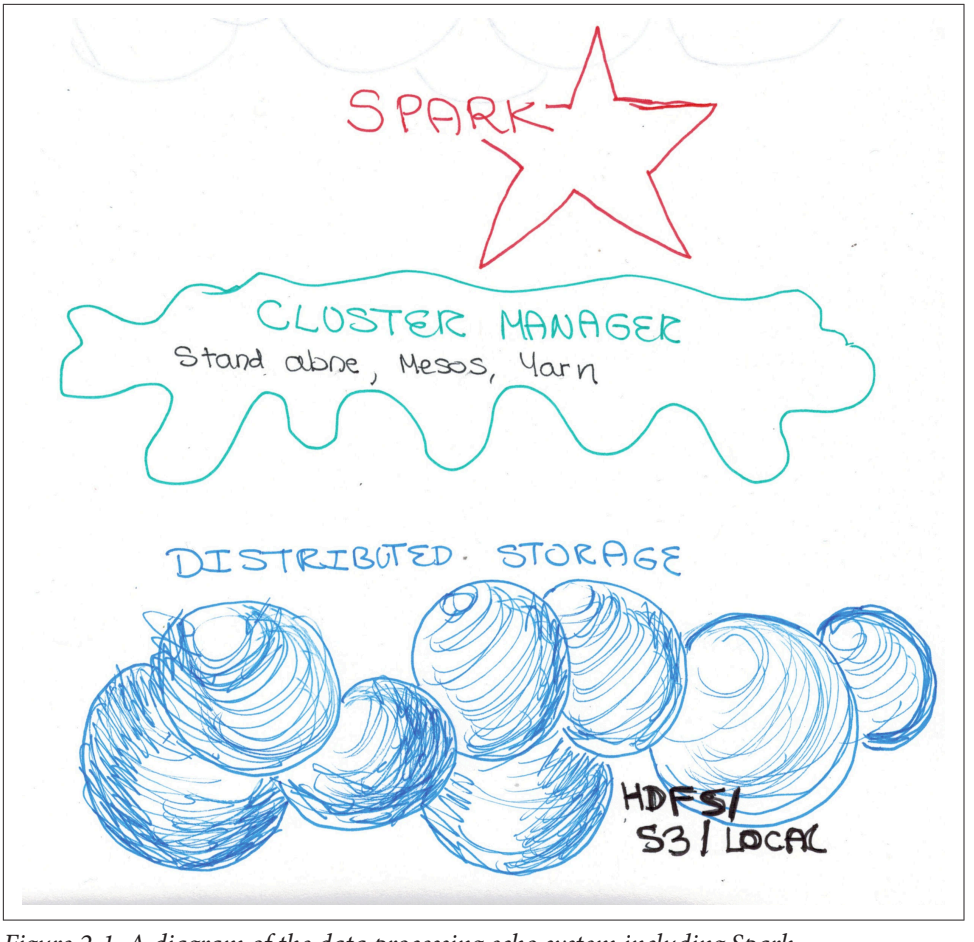


Figure 2-1. A diagram of the data processing echo system including Spark.

Spark Components

Spark provides a high-level query language to process data. Spark Core, the main data processing framework in the Spark ecosystem, has APIs in Scala, Java, and Python. Spark is built around a data abstraction called *Resilient Distributed Datasets* (RDDs). RDDs are a representation of lazily evaluated statically typed distributed collections. RDDs have a number of predefined "coarse grained" transformations (transformations that are applied to the entire dataset), such as `map`, `join`, and `reduce`, as well as I/O functionality, to move data in and out of storage or back to the driver.

In addition to Spark Core, the Spark ecosystem includes a number of other first-party components for more specific data processing tasks, including Spark SQL, Spark MLLib, Spark ML, and Graph X. These components have many of the same generic

performance considerations as the core. However, some of them have unique considerations - like SQL's different optimizer.

Spark SQL is a component that can be used in tandem with the Spark Core. Spark SQL defines an interface for a semi-structured data type, called `DataFrames` and a typed version called `Dataset`, with APIs in Scala, Java, and Python, as well as support for basic SQL queries. Spark SQL is a very important component for Spark performance, and much of what can be accomplished with Spark core can be applied to Spark SQL, so we cover it deeply in Chapter 3.

Spark has two machine learning packages, ML and MLlib. MLlib, one of Spark's machine learning components is a package of machine learning and statistics algorithms written with Spark. Spark ML is still in the early stages, but since Spark 1.2, it provides a higher-level API than MLlib that helps users create practical machine learning pipelines more easily. Spark MLLib is primarily built on top of RDDs, while ML is build on top of SparkSQL data frames. [4] Eventually the Spark community plans to move over to ML and deprecate MLlib. Spark ML and MLLib have some unique performance considerations, especially when working with large data sizes and caching, and we cover some these in ???.

Spark Streaming uses the scheduling of the Spark Core for streaming analytics on mini batches of data. Spark Streaming has a number of unique considerations such as the window sizes used for batches. We offer some tips for using Spark Streaming in ???.

Graph X is a graph processing framework built on top of Spark with an API for graph computations. Graph X is one of the least mature components of Spark, so we don't cover it in much detail. In future version of Spark, typed graph functionality will start to be introduced on top of the Dataset API. We will provide a cursory glance at Graph X in ???.

This book will focus on optimizing programs written with the Spark Core and Spark SQL. However, since MLLib and the other frameworks are written using the Spark API, this book will provide the tools you need to leverage those frameworks more efficiently. Who knows, maybe by the time you're done, you will be ready to start contributing your own functions to MLlib and ML!

Beyond first party components, a large number of libraries both extend Spark for different domains and offer tools to connect it to different data sources. Many libraries are listed at *http://spark-packages.org/*, and can be dynamically included at runtime with `spark-submit` or the `spark-shell` and added as build dependencies to our

4 See The MLlib documentation.

`maven` or `sbt` project. We first use Spark packages to add support for csv data in "Additional Formats" on page 66 and then in more detail in ???

Spark Model of Parallel Computing: RDDs

Spark allows users to write a program for the *driver* (or master node) on a cluster computing system that can perform operations on data in parallel. Spark represents large datasets as RDDs, immutable distributed collections of objects, which are stored in the *executors* or (slave nodes). The objects that comprise RDDs are called partitions and may be (but do not need to be) computed on different nodes of a distributed system. The Spark cluster manager handles starting and distributing the Spark executors across a distributed system according to the configuration parameters set by the Spark application. The Spark execution engine itself distributes data across the executors for a computation. See Figure 2-4.

Rather than evaluating each transformation as soon as specified by the driver program, Spark evaluates RDDs lazily, computing RDD transformations only when the final RDD data needs to be computed (often by writing out to storage or collecting an aggregate to the driver). Spark can keep an RDD loaded in memory on the executor nodes throughout the life of a Spark application for faster access in repeated computations. As they are implemented in Spark, RDDs are immutable, so transforming an RDD returns a new RDD rather than the existing one. As we will explore in this chapter, this paradigm of lazy evaluation, in memory storage and mutability allows Spark to be an easy-to-use as well as efficiently, fault-tolerant and general highly performant.

Lazy Evaluation

Many other systems for in-memory storage are based on "fine grained" updates to mutable objects, i.e., calls to a particular cell in a table by storing intermediate results. In contrast, evaluation of RDDs is completely lazy. Spark does not begin computing the partitions until and action is called. An action is a Spark operation which returns something other than an RDD, triggering evaluation of partitions and possibly returning some output to a non-Spark system for example bringing data back to the driver (with operations like `count` or `collect`) or writing data to external storage storage system (such as `copyToHadoop`). Actions trigger the scheduler, which builds a *directed acyclic graph* (called the DAG), based on the dependencies between RDD transformations. In other words, Spark evaluates an action by working backward to define the series of steps it has to take to produce each object in the final distributed dataset (each partition). Then, using this series of steps called the execution plan, the scheduler computes the missing partitions for each stage until it computes the whole RDD.

Performance & Usability Advantages of Lazy Evaluation

Lazy evaluation allows Spark to chain together operations that don't require communication with the driver (called transformations with one-to-one dependencies) to avoid doing multiple passes through the data. For example, suppose you have a program that calls a `map` and a `filter` function on the same RDD. Spark can look at each record once and compute both the `map` and the `filter` on each partition in the executor nodes, rather than doing two passes through the data, one for the `map` and one for the `filter`, theoretically reducing the computational complexity by half.

Spark's lazy evaluation paradigm is not only more efficient, it is also easier to implement the same logic in Spark than in a different framework like MapReduce, which requires the developer to do the work to consolidate her mapping operations. Spark's clever lazy evaluation strategy lets us be lazy and expresses the same logic in far fewer lines of code, because we can chain together operations with narrow dependencies and let the Spark evaluation engine do the work of consolidating them. Consider the classic word count example in which, given a dataset of documents, parses the text into words and then compute the count for each word. The word count example in MapReduce which is roughly fifty lines of code (excluding import statements) in Java compared to a program that provides the same functionality in Spark. A Spark implementation is roughly fifteen lines of code in Java and five in Scala. It can be found on the apache website. Furthermore if we were to filter out some "stop words" and punctuation from each document before computing the word count, this would require adding the filter logic to the mapper to avoid doing a second pass through the data. An implementation of this routine for MapReduce can be found here: *https://github.com/kite-sdk/kite/wiki/WordCount-Version-Three*. In contrast, we can modify the spark routine above by simply putting a filter step before we begin the code shown above and Spark's lazy evaluation will consolidate the map and filter steps for us.

Example 2-1.

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

Lazy Evaluation & Fault Tolerance

Spark is fault-tolerant, because each partition of the data contains the dependency information needed to re-calculate the partition. Distributed systems, based on mutable objects and strict evaluation paradigms, provide fault tolerance by logging updates or duplicating data across machines. In contrast, Spark does not need to maintain a log of updates to each RDD or log the actual intermediary steps, since the RDD itself contains all the dependency information needed to replicate each of its partitions. Thus, if a partition is lost, the RDD has enough information about its lineage to recompute it, and that computation can be parallelized to make recovery faster.

In Memory Storage and Memory Management

Spark's biggest performance advantage over MapReduce is in use cases involving repeated computations. Much of this performance increase is due to Spark's storage system. Rather than writing to disk between each pass through the data, Spark has the option of keeping the data on the executors loaded into memory. That way, the data on each partition is available in memory each time it needs to be accessed.

Spark offers three options for memory management: in memory deserialized data, in memory as serialized data, and on disk. Each has different space and time advantages.

1. *In memory as deserialized Java objects:* The most intuitive way to store objects in RDDs is as the deserialized Java objects that are defined by the driver program. This form of in memory storage is the fastest, since it reduces serialization time; however, it may not be the most memory efficient, since it requires the data to be as objects.
2. *As serialized data:* Using the Java serialization library, Spark objects are converted into streams of bytes as they are moved around the network. This approach may be slower, since serialized data is more CPU-intensive to read than deserialized data; however, it is often more memory efficient, since it allows the user to choose a more efficient representation for data than as Java objects and to use a faster and more compact serialization model, such as Kryo serialization. We will discuss this in detail in ???.
3. *On Disk:* Last, RDDs, whose partitions are too large to be stored in RAM on each of the executors, can be written to disk. This strategy is obviously slower for repeated computations, but can be more fault-tolerant for long strings of transformations and may be the only feasible option for enormous computations.

The `persist()` function in the RDD class lets the user control how the RDD is stored. By default, `persist()` stores an RDD as deserialized objects in memory, but the user can pass one of numerous storage options to the `persist()` function to control how the RDD is stored. We will cover the different options for RDD reuse in ???.

When persisting RDDs, the default implementation of RDDs evicts the least recently used partition (called LRU caching). However you can change this behavior and control Spark's memory prioritization with the `persistencePriority()` function in the RDD class. See ???.

Immutability and the RDD Interface

Spark defines an RDD interface with the properties that each type of RDD must implement. These properties include the RDD's dependencies and information about data locality that are needed for the execution engine to compute that RDD. Since RDDs are statically typed and immutable, calling a transformation on one RDD will not modify the original RDD but rather return a new RDD object with a new definition of the RDD's properties.

RDDs can be created in two ways: (1) by transforming an existing RDD or (2) from a Spark Context, `SparkContext`. The Spark Context represents the connection to a Spark cluster and one running Spark application. The Spark Context can be used to create an RDD from a local Scala object (using the `makeRDD`, or `parallelize` methods) or by reading from stable storage (text files, binary files, a Hadoop Context, or a Hadoop file). DataFrames can be created by the Spark SQL equivalent to a Spark Context, `SQLContext` object, which can be created from a Spark Context.

Spark uses five main properties to represent an RDD internally. The three required properties are the list of partition objects, a function for computing an iterator of each partition, and a list of dependencies on other RDDs. Optionally, RDDs also include a partitioner (for RDDs of rows of key-value pairs represented as Scala tuples) and a list of preferred locations (for the HDFS file). Although, as an end user, you will rarely need these five properties and are more likely to use predefined RDD transformations, it is helpful to understand the properties and know how to access them for conceptualizing RDDs and for debugging. These five properties correspond to the following five methods available to the end user (you):

- `partitions()` Returns an array of the partition objects that make up the parts of the distributed dataset. In the case of an RDD with a partitioner, the value of the index of each partition will correspond to the value of the getPartition function for each key in the data associated with that partition.
- `iterator(p, parentIters)` Computes the elements of partition p given iterators for each of its parent partitions. This function is called in order to compute each of the partitions in this RDD. This is not intended to be called directly by the user, rather this is used by Spark when computing actions. Still, referencing the implementation of this function can be useful in determining how each partition of an RDD transformation is evaluated.

- `dependencies()` Returns a sequence of dependency objects. The dependencies let the scheduler know how this RDD depends on other RDDs. There are two kinds of dependencies: *Narrow Dependencies* (`NarrowDependency` objects), which represent partitions that depend on one or a small subset of partitions in the parent, and *Wide Dependencies* (`ShuffleDependency` objects), which are used when a partition can only be computed by rearranging all the data in the parent. We will discuss the types of dependencies in "Wide vs. Narrow Dependencies" on page 26.
- `partitioner()` Returns a Scala option type that contains a `partitioner` object if the RDD has a function between datapoint and partitioner associated with it, such as a `hashPartitioner`. This function returns `None` for all RDDs that are not of type tuple (do not represent Key-value data). An RDD that represents an HDFS file (implemented in `NewHadoopRDD.scala`) has a partitioner for each block of the file. We will discuss partitioning in detail in ???.
- `preferredLocations(p)` Returns information about the data locality of a partition, p. Specifically, this function returns a sequence of strings representing some information about each of the nodes, where the split `p` is stored. In an RDD representing an HDFS file, each string in the result of `preferedLocations` is the Hadoop name of the node.

Types of RDDs

In practice, the Scala API contains an abstract class, `RDD`, which contains not only the five core functions of RDDs, but also those transformations and actions that are available to all RDDs, such as `map` and `collect`. Functions defined only on RDDs of a particular type are defined in several RDD Functions classes, including `PairRDDFunctions`, `OrderedRDDFunctions` and `GroupedRDDFunctions`. The additional methods in these classes are made available by implicit conversion from the abstract RDD class, based on type information or when a transformation is applied to an RDD.

The Spark API also contains specific implementations of the RDD class that define specific behavior by overriding the core properties of the RDD. These include the `NewHadoopRDD` class discussed above, which represents an RDD created from an HDFS file system, and `ShuffledRDD`, which represents an RDD, that was already partitioned. Each of these RDD implementations contains functionality that is specific to RDDs of that type. Creating an RDD, either through a transformation or from a Spark Context, will return one of these implementations of the RDD class. Some RDD operations have a different signature in Java than in Scala. These are defined in the `JavaRDD.java` class.

We will discuss the different types of RDDs and RDD transformations in detail in ??? and ???

Functions on RDDs: Transformations vs. Actions

There are two types of functions defined on RDDs, *actions* and *transformations*. Actions are functions that return something that isn't an RDD and transformations that return another RDD.

Each Spark program must contain an action, since actions either bring information back to the driver or write the data to stable storage. Actions that bring data back to the driver include `collect`, `count`, `collectAsMap`, `sample`, `reduce` and `take`.

Some of these actions do not scale well, since they can cause memory errors in the driver. In general, it is best to use actions like `take`, `count`, and `reduce`, which bring back a fixed amount of data to the driver, rather than `collect` or `sample`.

Actions that write to storage include `saveAsTextFile`, `saveAsSequenceFile`, and `saveAsObjectFile`. Actions that save to Hadoop are made available only on RDDs of key-value pairs; they are defined both in the `PairRDDFunctions` class (which provides methods for RDDs or tuple type by implicit conversation) and the `NewHadoopRDD` class, which is an implementation for RDDs that were created by reading from Hadoop. Functions that return nothing (unit type and Scala), such as `foreach`, are also actions and force execution of a Spark job that can be used to write out to other data sources or any other arbitrary action.

Most of the power of the Spark API is in its transformations. Spark transformations are general coarse grained transformations used to sort, reduce, group, sample, filter, and map distributed data. We will talk about transformations in detail in both ???, which deals exclusively with transformations on RDDs of key / value data, and ???, and we will talk about advanced performance considerations with respect to data transformations.

Wide vs. Narrow Dependencies

For the purpose of understanding how RDDs are evaluated, the most important thing to know about transformations is that they fall into two categories: transformations with *narrow dependencies* and transformations with *wide dependencies*. The narrow vs. wide distinction has significant implications for the way Spark evaluates a transformation and, consequently, for its performance. We will define narrow and wide transformations for the purpose of understanding Spark's execution paradigm in "Spark Job Scheduling" on page 28 of this chapter, but we will save the longer explanation of the performance considerations associated with them for ???.

Conceptually, narrow transformations are operations with dependencies on just one or a known set of partitions in the parent RDD which can be determined at design

time. Thus narrow transformations can be executed on an arbitrary subset of the data without any information about the other partitions. In contrast, transformations with wide dependencies cannot be executed on arbitrary rows and instead require the data to be partitioned in a particular way. Transformations with wide dependencies include, `sort`, `reduceByKey`, `groupByKey`, `join`, and anything that calls for repartition.

We call the process of moving the records in an RDD to accommodate a partitioning requirement, a *shuffle*. In certain instances, for example, when Spark already knows the data is partitioned in a certain way, operations with wide dependencies do not cause a shuffle. If an operation will require a shuffle to be executed, Spark adds a `ShuffledDependency` object to the dependency list associated with the RDD. In general, shuffles are expensive, and they become more expensive the more data we have, and the greater percentage of that data has to be moved to a new partition during the shuffle. As we will discuss at length in ???, we can get a lot of performance gains out of Spark programs by doing fewer and less expensive shuffles.

The next two diagrams illustrates the difference in the dependency graph for transformations with narrow vs. transformations with wide dependencies. On top, are narrow dependencies in which each child partition (each of the blue squares on the bottom rows) depends on a known subset of parent partitions (narrow dependencies are shown with blue arrows). The left represents a dependency graph of narrow transformations such as `map`, `filter`, `mapPartitions` and `flatMap`. On the upper right are dependencies between partitions for `coalesce`, a narrow transformation. In this instance we try to illustrate that the child partitions may depend on multiple parent partitions, but that so long as the set of parent partitions can be determined regardless of the values of the data in the partitions, the transformation qualifies as narrow.

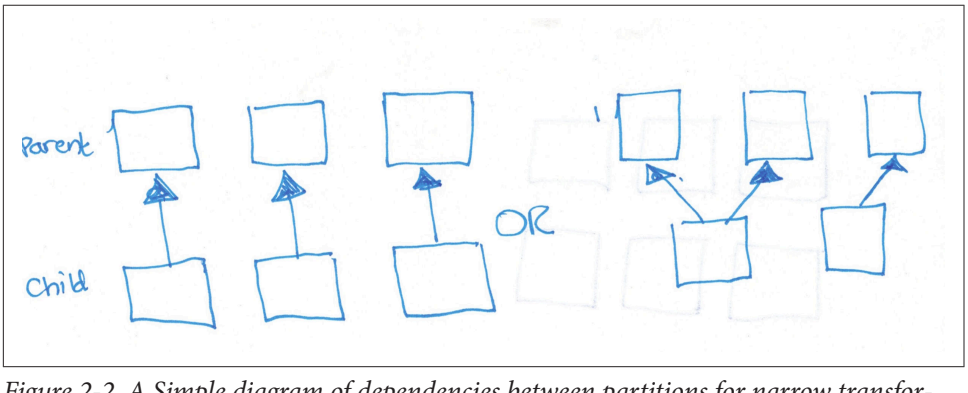


Figure 2-2. A Simple diagram of dependencies between partitions for narrow transformations.

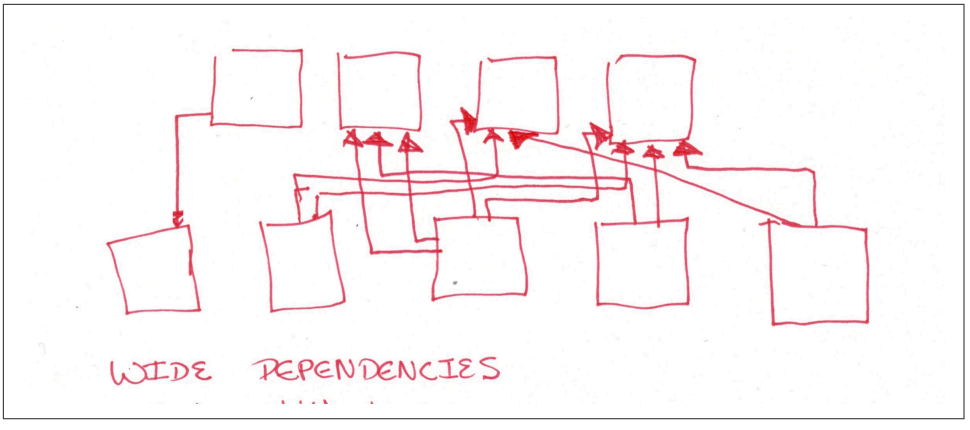


Figure 2-3. A Simple diagram of dependencies between partitions for wide transformations.

The second diagram shows wide dependencies between partitions. In this case the child partitions (shown below) depend on an arbitrary set of parent partitions. The wide dependencies (displayed as red arrows) cannot be known fully before the data is evaluated. In contrast to the `coalesce` operation, data is partitioned according to its value. The dependency graph for any operations that cause a shuffle such as `groupBy Key`, `reduceByKey`, `sort`, and `sortByKey` follows this pattern.

Join is a bit more complicated, since it can have wide or narrow dependencies depending on how the two parent RDDs are partitioned. We illustrate the dependencies in different scenarios for the join operation in "Core Spark Joins" on page 79.

Spark Job Scheduling

A Spark application consists of a driver process, which is where the high-level Spark logic is written, and a series of executor processes that can be scattered across the nodes of a cluster. The Spark program itself runs in the driver node and parts are sent to the executors. One Spark cluster can run several Spark applications concurrently. The applications are scheduled by the cluster manager and correspond to one SparkContext. Spark applications can run multiple concurrent jobs. Jobs correspond to each action called on an RDD in a given application. In this section, we will describe the Spark application and how it launches Spark jobs, the processes that compute RDD transformations.

Resource Allocation Across Applications

Spark offers two ways of allocating resources across applications: *static allocation* and *dynamic allocation*. With static allocation, each application is allotted a finite maxi-

mum of resources on the cluster and reserves them for the duration of the application (as long as the Spark Context is still running). Within the static allocation category, there are many kinds of resource allocation available, depending on the cluster. For more information, see the Spark documentation for like::http://spark.apache.org/docs/latest/job-scheduling.html[job scheduling].

Since 1.2, Spark offers the option of dynamic resource allocation which expands the functionality of static allocation. In dynamic allocation, executors are added and removed from a Spark application as needed, based on a set of heuristics for estimated resource requirement. We will discuss resource allocation in ???.

The Spark application

A Spark application corresponds to a set of Spark jobs defined by one Spark Context in the driver program. A Spark application begins when a Spark Context is started. When the Spark Context is started, each worker node starts an executor (its own Java Virtual Machine, JVM).

The Spark Context determines how many resources are allotted to each executor, and when a Spark job is launched, each executor has slots for running the tasks needed to compute an RDD. In this way, we can think of one Spark Context as one set of configuration parameters for running Spark jobs. These parameters are exposed in the `SparkConf` object, which is used to create a Spark Context. We will discus how to use the parameters in ???. Often, but not always, applications correspond to users. That is, each Spark program running on your cluster likely uses one Spark Context.

RDDs cannot be shared between applications, so transformations, such as `join` that use more than one RDD must have the same Spark Context.

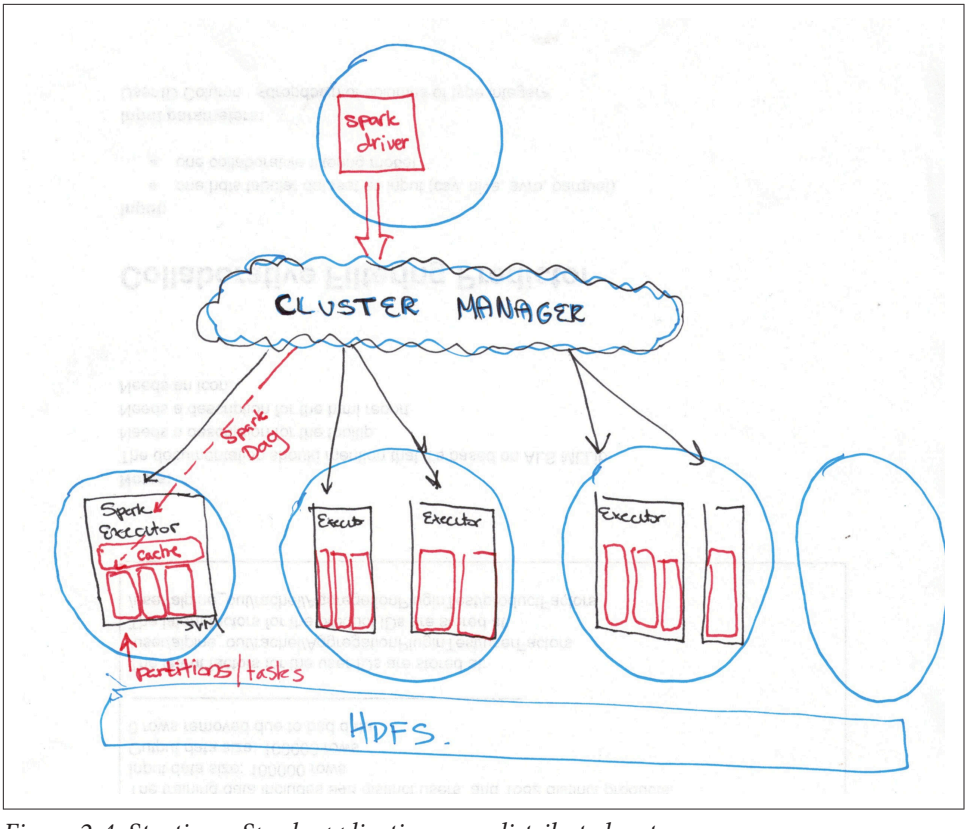


Figure 2-4. Starting a Spark application on a distributed system.

The above diagram illustrates what happens when we start a Spark Context. First The driver program pings the cluster manager. The cluster manager launches a number of Spark executors, JVMs, (shown as black boxes) on the worker nodes of the cluster (shown as blue circles). One node can have multiple Spark executors, but an executor cannot span multiple nodes. An RDD will be evaluated across the executors in partitions (shown as red rectangles). Each executor can have multiple partitions, but a partition cannot be spread across multiple executors.

By default, Spark queues jobs in a first in, first out basis. However, Spark does offer a fair scheduler, which assigns tasks to concurrent jobs in round-robin fashion, i.e. parceling out a few tasks for each job until the jobs are all complete. The fair scheduler ensures that jobs get a more even share of cluster resources. The Spark application then launches jobs in the order that their corresponding actions were called on the Spark Context.

The Anatomy of a Spark Job

In the Spark lazy evaluation paradigm, a Spark application doesn't "do anything" until the driver program calls an action. With each action, the Spark scheduler builds an execution graph and launches a *Spark job*. Each job consists of *stages*, which are steps in the transformation of the data needed to materialize the final RDD. Each stage consists of a collection of *tasks* that represent each parallel computation and are performed on the executors.

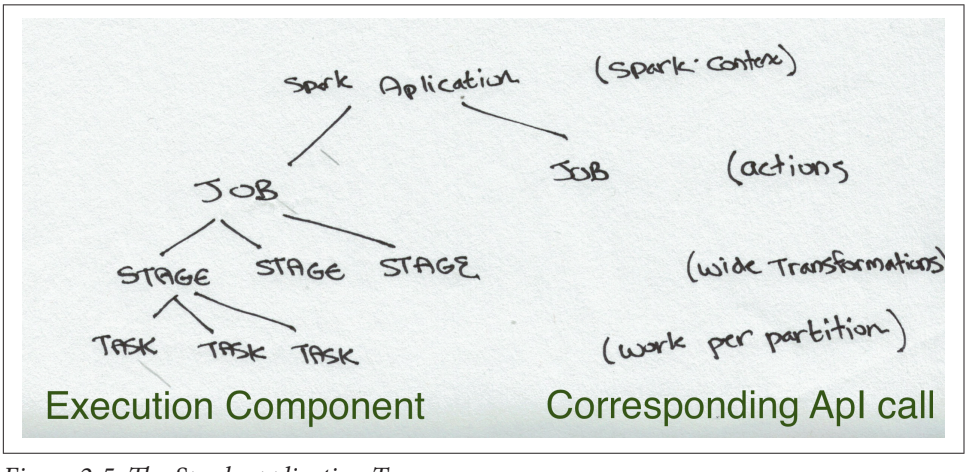


Figure 2-5. The Spark application Tree

The above diagram shows a tree of the different components of a Spark application. An application corresponds to starting a Spark Context. Each application may contain many jobs that correspond to one RDD action. Each job may contain several stages which correspond to each wide transformation. Each stage is composed of one or many tasks which correspond to a parallelizable unit of computation done in each stage. There is one task for each partition in the resulting RDD of that stage.

The DAG

Spark's high-level scheduling layer uses RDD dependencies to build a *Directed Acyclic Graph* (a DAG) of stages for each Spark job. In the Spark API, this is called the `DAG Scheduler`, and as you have probably noticed, errors that have to do with connecting to your cluster, your configuration parameters, or launching a Spark job show up as DAG Scheduler errors. This is because the execution of a Spark job is handled by the DAG. The DAG builds a graph of stages for each job, determines the locations to run each task, and passes that information on to the `TaskScheduler`, which is responsible for running tasks on the cluster.

Jobs

A Spark job is the highest element of Spark's execution hierarchy. Each Spark job corresponds to one action, called by the Spark application. As we discussed in "Functions on RDDs: Transformations vs. Actions" on page 26, one way to conceptualize an action is as something that brings data out of the RDD world of Spark into some other storage system (usually by bringing data to the driver or some stable storage system).

Since the edges of the Spark execution graph are based on dependencies between RDD transformations (as illustrated by Figure 2-2 and Figure 2-3), an operation that returns something other than an RDD cannot have any children. Thus, an arbitrarily large set of transformations may be associated with one execution graph. However, as soon as an action is called, Spark can no longer add to that graph and launches a job including those transformations that were needed to evaluate the final RDD that called the action.

Stages

Recall that Spark lazily evaluates transformations; transformations are not executed until actions are called. As mentioned above, a job is defined by calling an action. The action may include several transformations, and wide transformations define the breakdown of jobs into *stages*.

Each stage corresponds to a `ShuffleDependency` created by a wide transformation in the Spark program. At a high level, one stage can be thought of as the set of computations (tasks) that can each be computed on one executor without communication with other executors or with the driver. In other words, a new stage begins each time in the series of steps needed to compute the final RDD that data has to be moved across the network. These dependencies that create stage boundaries are called ShuffleDependencies, and as we discussed in "Wide vs. Narrow Dependencies" on page 26, they are caused by those wide transformations, such as `sort` or `groupByKey`, which require the data to be re-distributed across the partitions. Several transformations with narrow dependencies can be grouped into one stage. For example, a `map` and a `filter` step are combined into one stage, since neither of these transformations require a shuffle; each executor can apply the `map` and `filter` steps consecutively in one pass of the data.

Because the stage boundaries require communication with the driver, the stages associated with one job generally have to be executed in sequence rather than in parallel. It is possible to executed stages in parallel if they are used to compute different RDDs which are combined in a downstream transformation such as a join. However, the wide transformations needed to compute one RDD have to be computed in sequence and thus, it is usually desirable to design your program to require fewer shuffles.

Tasks

A stage consists of tasks. The *task* is the smallest unit in the execution hierarchy, and each can represent one local computation. Each of the tasks in one stage all execute the same code on a different piece of the data. One task cannot be executed on more than one executor. However, each executor has a dynamically allocated number of slots for running tasks and may run many tasks concurrently throughout its lifetime. The number of tasks per stage corresponds to the number of partitions in the output RDD of that stage.

The following diagram shows the evaluation of a Spark job that is the result of a driver program that calls the following simple Spark program:

Example 2-2.

```
def simpleSparkProgram(rdd : RDD[Double]): Long ={
//stage1
  rdd.filter(_< 1000.0)
     .map(x => (x , x) )
//stage2
     .groupByKey()
     .map{ case(value, groups) => (groups.sum, value)}
//stage 3
     .sortByKey()
     .count()
}
```

The stages (black boxes) are bounded by the shuffle operations `groupByKey` and `sortByKey`. Each stage consists of several tasks, one for each partition in the result of the RDD transformations (shown as red squares), which are executed in parallel.

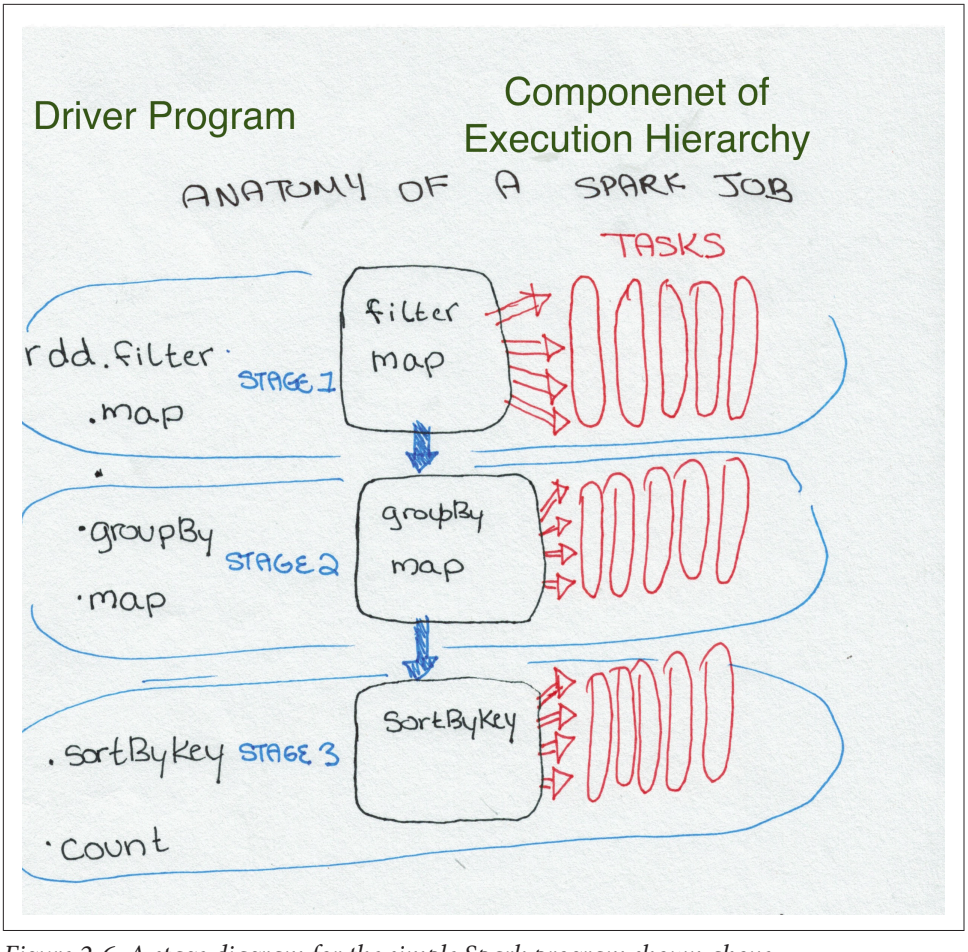


Figure 2-6. A stage diagram for the simple Spark program shown above.

In some ways, the simplest way to think of the Spark execution model is that a Spark job is the set of RDD transformations needed to compute one final result. Each stage corresponds to a segment of work, which can be accomplished without involving the driver. In other words, one stage can be computed without moving data across the partitions. Within one stage, the tasks are the units of work done for each partition of the data.

Conclusion

Spark has an innovative, efficient model of parallel computing centering on lazily evaluated, immutable, distributed datasets, RDDs. Spark exposes RDDs as an interface, and RDD methods can be used without any knowledge of their implementation.

Because of Spark's ability to run jobs concurrently, to compute jobs across multiple nodes, and to materialize RDDs lazily, the performance implications of similar logical patterns may differ widely and errors may surface from misleading places. Thus, it is important to understand how the execution model for your code is assembled in order to write and debug Spark code. Furthermore, it is often possible to accomplish the same tasks in many different ways using the Spark API, and a strong understanding of how your code is evaluated will help you optimize its performance. In this book, we will focus on ways to design Spark applications to minimize network traffic, memory errors, and the cost of failures.

CHAPTER 3

DataFrames, Datasets & Spark SQL

Spark SQL and its DataFrames and Datasets interfaces are the future of Spark performance, with more efficient storage options, advanced optimizer, and direct operations on serialized data. These components are super important for getting the best of Spark performance - see Figure 3-1.

Figure 3-1. Spark SQL Performance Relative Simple RDDs From SimplePerfTest.scala Aggregating avg Fuzziness

These are relatively new components; Datasets was introduced in Spark 1.6, DataFrames in Spark 1.3, and the SQL engine in Spark 1.0. This chapter is focused on

helping you learn how to best use Spark SQL's tools. For tuning params a good follow up is ???.

Spark's DataFrames have very different functionality compared to traditional DataFrames like Panda's and R. While these all deal with structured data, it is important not to depend on your existing intuition surrounding DataFrames.

Like RDDs, DataFrames and Datasets represent distributed collections, with additional schema information not found in RDDs. This additional schema information is used to provide a more efficient storage layer and in the optimizer. Beyond schema information, the operations performed on DataFrames are such that the optimizer can inspect the logical meaning rather than arbitrary functions. Datasets are an extension of DataFrames bringing strong types, like with RDDs, for Scala/Java and more RDD-like functionality within the Spark SQL optimizer. Compared to working with RDDs, DataFrames allow Spark's optimizer to better understand our code and our data, which allows for a new class of optimizations we explore in "Query Optimizer" on page 75.

While Spark SQL, DataFrames, and Datasets provide many excellent enhancements, they still have some rough edges compared to traditional processing with "regular" RDDs. The Dataset API, being brand new at the time of this writing, is likely to experience some changes in future versions.

Getting Started with the HiveContext (or SQLContext)

Much as the `SparkContext` is the entry point for all Spark applications, and the `StreamingContext` is for all streaming applications, the `HiveContext` and `SQLContext` serve as the entry points for Spark SQL. The names of these entry points can be a bit confusing, and it is important to note the `HiveContext` **does not require** a Hive installation. The primary reason to use the `SQLContext` is if you have conflicts with the Hive dependencies that cannot be resolved. The `HiveContext` has a more complete SQL parser as well as additional user defined functions (UDFs) [5] and should be used whenever possible.

Like with all of the Spark components, you need to import a few extra components as shown in Example 3-1. If you are using the `HiveContext` you should import those

5 UDFs allow us to extend SQL to have additional powers, such as computing the geo-spatial distance between points

components as well. Once you have imported the basics, you can create the entry point as in Example 3-2.

Example 3-1. Spark SQL imports

```
import org.apache.spark.sql.{DataFrame, SQLContext, Row}
import org.apache.spark.sql.catalyst.expressions.aggregate._
import org.apache.spark.sql.expressions._
import org.apache.spark.sql.functions._
// Additional imports for using HiveContext
import org.apache.spark.sql.hive._
import org.apache.spark.sql.hive.thriftserver._
```

Example 3-2. Creating the HiveContext

```
    val hiveContext = new HiveContext(sc)
    // Import the implicits, unlike in core Spark the implicits are defined on the context
    import hiveContext.implicits._
```

To use the `HiveContext` you will need to add both Spark's `SQL` and `Hive` components to your dependencies. If you are using the sbt-spark-package plugin you can do this by just adding `SQL` and `Hive` to your list of `sparkComponents`.

Example 3-3. Add Spark SQL & Hive component to sbt-spark-package build

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

For maven compatible build systems, the coordinates for Spark's SQL and Hive components in 1.6.0 are `org.apache.spark:spark-sql_2.10:1.6.0` and `org.apache.spark:spark-hive_2.10:1.6.0`.

Example 3-4. Add Spark SQL & Hive component to "regular" sbt build

```
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-sql" % "1.6.0",
  "org.apache.spark" %% "spark-hive" % "1.6.0")
```

Example 3-5. Add Spark SQL & Hive component to maven pom file

```
 <dependency> <!-- Spark dependency -->
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.10</artifactId>
  <version>1.6.0</version>
</dependency>
<dependency> <!-- Spark dependency -->
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-hive_2.10</artifactId>
```

```
  <version>1.6.0</version>
</dependency>
```

While it's not required, if you do have an existing Hive Metastore to which you wish to connect your `HiveContext`, you can copy your `hive-site.xml` to Spark's `conf/` directory.

As of Spark 1.6.0, the default Hive Metastore version is 1.2.1. For other versions of the Hive Metastore you will need to set the `spark.sql.hive.metastore.version` property to the desired versions as well as set `spark.sql.hive.metastore.jars` to either "maven" (to have Spark retrieve the jars) or the system path where the Hive JARs are present.

If you can't include the Hive dependencies with our application, you can leave out Spark's Hive component and instead create a `SQLContext`. This provides much of the same functionality, but uses a less capable SQL parser and lacks certain Hive based user defined functions (UDFs) and user dfined aggregate functions (UDAFs).

Example 3-6. Creating the SQLContext

```
val sqlContext = new SQLContext(sc)
// Import the implicits, unlike in core Spark the implicits are defined on the context
import sqlContext.implicits._
```

If you are using the Spark Shell you will automatically get a SQLContext. This SQLContext will be a HiveContext if Spark was built with Hive support (e.g. `-PHive`).

As with the core Spark Context and Streaming Context, the Hive/SQLContext is used to load your data. JSON format data is widely popular, and some sample data about pandas, as well as some complex data for the Goldilocks example, is included in this book's sample `resources` directory. JSON is especially interesting since it lacks schema information, and Spark needs to do some work to infer the schema from our data. We will cover the full loading and saving API for JSON in "JSON" on page 59, but to get started let's load a sample we can use to explore the schema.

Example 3-7. Load JSON sample

```
val df = sqlCtx.read.json(path)
```

Basics of Schemas

The schema information, and the optimizations it enables, is one of the core differences between Spark SQL and core Spark. Inspecting the schema is especially useful for DataFrames since you don't have the templated type you do with RDDs or Datasets. Schemas are normally handled automatically by Spark SQL, either inferred when loading the data or computed based on the parent DataFrames and the transformation being applied.

DataFrames expose the schema in both human-readable or programmatic formats. `printSchema()` will show us the schema of a DataFrame and is most commonly used when working in the shell to figure out with what you are working. This is especially useful for data formats, like in JSON, where the schema may not be immediately visible by looking at only a few records or reading a header. For programmatic usage, you can get the schema by simply calling `schema`, which is often used in ML pipeline transformers. Since you are likely familiar with case classes and JSON, let's examine how the equivalent Spark SQL schema would be represented.

Example 3-8. JSON data which would result in an equivalent schema

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

Example 3-9. Equivalent case class to the next two examples

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

Example 3-10. StructField case class

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

Example 3-11. Sample schema information for nested structure (schema) - manually formatted

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

```
StructField(happy,BooleanType,false),
StructField(attributes,ArrayType(DoubleType,false),true)),true),true))
```

Example 3-12. Sample Schema Information for Nested Structure (printSchema)

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

From here we can dive into what this schema information means and look at how to construct more complex schemas. The first part is a `StructType` which contains a list of fields. It's important to note you can nest `StructTypes`, like how a case class can contain additional case classes. The fields in the `StructType` are defined with `StructField` which specifies the name, type (see Table 3-1 and Table 3-2 for a listing of types), and a boolean indicating if the field may be null/missing.

Table 3-1. Basic Spark SQL types

Scala Type	SQL Type	Details
Byte	ByteType	1 byte signed integers (-128,127)
Short	ShortType	2 byte signed integers (-32768,32767)
Int	IntegerType	4 byte signed integers (-2147483648,2147483647)
Long	LongType	8 byte signed integers (-9223372036854775808,9223372036854775807)
java.math.BigDecimal	DecimalType	Arbitrary precision signed decimals
Float	FloatType	4 byte floating point number
Double	DoubleType	8 byte floating point number
Array[Byte]	BinaryTpe	Array of bytes
Boolean	BooleanTpe	true/false
java.sql.Date	DateType	Date without time information
java.sql.Timestamp	TimestampType	Date with time information (second precision)

Scala Type	SQL Type	Details
String	StringType	Character string values (stored as UTF8)

Table 3-2. Complex Spark SQL types

Scala Type	SQL Type	Details	Example
Array[T]	ArrayType(elementType, containsNull)	Array of single type of element, containsNull true if any null elements.	`Array[Int] => ArrayType(IntegerType, true)`
Map[K, V]	MapType(elementType, valueType, valueContainsNull)	Key/value map, valueContainsNull if any values are null.	`Map[String, Int] => MapType(StringType, IntegerType, true)`
case class	StructType(`List[StructFields]`)	Named fields of possible heterogeneous types, similar to a case class or JavaBean.	`case class Panda(name: String, age: Int) => StructType(List(StructField("name", StringType, true), StructField("age", IntegerType, true)))`

As you saw in Example 3-11, you can nest StructFields and all of the complex Spark SQL types.

Now that you've got an idea of how to understand and, if needed, construct schemas for, your data, you are ready to start exploring the DataFrame interfaces.

Spark SQL schemas are eagerly evaluated, unlike the data underneath. If you find yourself in the shell and uncertain of what a transformation will do - try it and print the schema. See Example 3-12.

DataFrame API

Spark SQL's DataFrame API allows us to work with DataFrames without having to register temporary tables or generate SQL expressions. The DataFrame API has both transformations and actions. The transformations on DataFrames are more relational in nature, with the Dataset API (covered next) offering a more functional-style API.

Transformations

Transformations on DataFrames are similar in concept to RDD transformations, but with a more relational flavor. Instead of specifying arbitrary functions, which the optimizer is unable to introspect, you use a restricted expression syntax so the optimizer can have more information. As with RDDs, we can broadly break down transformations into single, multi, key/value, and grouped/windowed transformations.

Spark SQL transformations are only partially lazy; the schema is eagerly evaluated.

Simple DataFrame Transformations & SQL Expressions

Simple DataFrame transformations allow us to do most of the standard things one can do when working a row at a time.[5] You can still do many of the same operations defined on RDDs, except using Spark SQL expressions instead of arbitrary functions. To illustrate this we will start by examining the different kinds of filter operations available on DataFrames.

DataFrame functions, like `filter` accept Spark SQL expressions instead of lambdas. These expressions allow the optimizer to understand what the condition represents, and with filter, it can often be used to skip reading unnecessary records.

To get started, let's look at a SQL expression to filter our data for unhappy pandas using our existing schema. The first step is looking up the column that contains this information. In our case it is "happy", and for our DataFrame (called df) we access the column through the apply function (e.g. (`df("happy")`)). The `filter` expression requires the expression to return a boolean value, and if you wanted to select happy pandas, the entire expression could be retrieving the column value. However, since we want to find the unhappy pandas though, we can check to see that happy isn't true using the `!==` operator as shown in Example 3-13.

Example 3-13. Simple filter for unhappy pandas

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

5 A row at a time allows for narrow transformations with no shuffle.

To lookup op the column, we can either provide the column name on the specific DataFrame or use the implicit `$` operator for column lookup. This is especially useful when the DataFrame is anonymous. The `!` binary negation function can be used together with `$` to simplify our expression from Example 3-13 down to `df.filter(!$("happy"))`.

This illustrates how to access a specific column from a DataFrame. For accessing other structures inside of DataFrames, like nested structs, keyed maps, and array elements, use the same apply syntax. So, if the first element in the attributes array represent squishiness, and you only want very squishy pandas, you can access that element by writing `df("attributes")(0) >= 0.5`.

Our expressions need not be limited to a single column. You can compare multiple columns in our "filter" expression. Complex filters like this are more difficult to push down to the storage layer, so you may not see the same speedup over RDDs that you see with simpler filters.

Example 3-14. More complex filter

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

Spark SQL's column operators are defined on the column class, so a filter containing the expression `0 >= df.col("friends")` will not compile since Scala will use the `>=` defined on 0. Instead you would write `df.col("friend") <= 0` or convert 0 to a column literal with `lit` [5].

Spark SQLs DataFrame API has a very large set of operators available. You can use all of the standard mathematical operators on floating points, along with the standard logical and bitwise operations (prefix with `bitwise` to distinguish from logical). Columns use `===` and `!==` for equality to avoid conflict with Scala internals. For columns of strings, `startsWith/endsWith`, `substr`, `like`, and `isNull` are all available. The full set of operations is listed in org.apache.spark.sql.Column and covered in Table 3-3 and Table 3-4.

5 A column literal is a column with a fixed value that doesn't change between rows (i.e. constant).

Table 3-3. Spark SQL Scala operators

Scala Operator	Java Equivalent	Input Column Types	Output Type	Purpose	Sample	Result
`!==`	`notEqual`	Any	Boolean	Check if expressions not equal	"hi" !== "bye"	true
`%`	`mod`	Numeric	Numeric	Modulo	`10 % 5`	0
`&&`	`and`	Boolean	Boolean	Boolean and	`true && false`	`false`
`*`	`multiply`	Numeric	Numeric	Multiply expressions	`2 * 21`	42
`+`	`plus`	Numeric	Numeric	Sum expression	`2 + 2`	4
`-`	`minus`	Numeric	Numeric	Subtraction	`2 - 2`	0
`-`	`unary_-`	Numeric	Numeric	Unary subtraction	`-42`	`-42`
`/`	`division`	Numeric	Double	Division	`43/2`	`21.5`
`<`	`lt`	Comparable	Boolean	Less than	`"a" < "b"`	`true`
`<=`	`leq`	Comparable	Boolean	Less than or equal to	`"a" <= "a"`	`true`
`===`	`equals`	Any	Any	Equality test (unsafe on null values)	`"a" === "a"`	`true`
`<=>`	`eqNullSafe`	Any	Any	Equality test (safe on null values)	`"a" <=> "a"`	`true`
`>`	`gt`	Comparable	Boolean	Greater than	`"a" > "b"`	`false`
`>=`	`gt`	Comparable	Boolean	Greater than or equal to	`"a" >= "b"`	`false`

Table 3-4. Spark SQL expression operators

Operator	Input Column Types	Output Type	Purpose	Sample	Result
`apply`	Complex types	Type of field accessed	Get value from complex type (e.g. structfield/map lookup or array index)	`[1,2,3].apply(0)`	1
`bitwiseAND`	Integral Type [a]	Same as input	Computes and bitwise	`21.bitwiseAND(11)`	1
`bitwiseOR`	Integral Type [a]	Same as input	Computes or bitwise	`21.bitwiseOR(11)`	31

Operator	Input Column Types	Output Type	Purpose	Sample	Result
`bitwiseXOR`	Integral Type [a]	Same as input	Computes bitwise exclusive or	`21.bitwiseXOR(11)`	`30`

[a] Integral types include ByteType, IntegerType, LongType, and ShortType.

Not all Spark SQL expressions can be used in every API call. For example, Spark SQL joins do not support complex operations, and filter requires that the expression result in a boolean, and similar.

In addition to the operators directly specified on the column, an even larger set of functions on columns exists in org.apache.spark.sql.functions, some of which we cover in tables Table 3-5 and Table 3-6. For illustration, this example shows the values for each column at a specific row, but keep in mind that, these functions are called on columns not values.

Table 3-5. Spark SQL standard functions

Function name	Purpose	Input Types	Example usage	Result
`lit(value)`	Convert a Scala symbol to a column literal [3]	Column & Symbol	`lit(1)`	`Column(1)`
`array`	Create a new array column	Must all have the same Spark SQL type	`array(lit(1),lit(2))`	array(1,2)
`isNaN`	Check if not a number	Numeric	`isNan(lit(100.0))`	false
`not`	Opposite value	Boolean	`not(lit(true))`	false

Table 3-6. Spark SQL Common Mathematical Expressions

Function name	Purpose	Input Types	Example usage	Result
`abs`	Absolute value	Numeric	`abs(lit(-1))`	1
`sqrt`	Square root	Numeric	`sqrt(lit(4))`	2
`acos`	Inverse cosine	Numeric	`acos(lit(0.5))`	1.04…. [a]
`asin`	Inverse sine	Numeric	`asin(lit(0.5))`	0.523… [a]
`atan`	Inverse tangent	Numeric	`atan(lit(0.5))`	0.46… [a]

Function name	Purpose	Input Types	Example usage	Result
`cbrt`	Cube root	Numeric	`sqrt(lit(8))`	2
`ceil`	Ceiling	Numeric	`ceil(lit(8.5))`	9
`cos`	Cosine	Numeric	`cos(lit(0.5))`	0.877…. [a]
`sin`	Sine	Numeric	`sin(lit(0.5))`	0.479… [a]
`tan`	Tangent	Numeric	`tan(lit(0.5))`	0.546… [a]
`exp`	Exponent	Numeric	`exp(lit(1.0))`	2.718… [a]
`floor`	Ceiling	Numeric	`floor(lit(8.5))`	8
`least`	Minimum value	Numerics	`least(lit(1), lit(-10))`	-10

[a] Truncated for display purposes.

Table 3-7. Functions for use on Spark SQL arrays

Function name	Purpose	Example usage	Result
`array_contains`	If an array contains a value	`array_contains(lit(Array(2,3,-1), 3))`	true
`sort_array`	Sort an array (ascending default)	`sort_array(lit(Array(2,3,-1)))`	`Array(-1,2,3)`
`explode`	Create a row for each element in the array - often useful when working with nested JSON records. Either takes a column name or additional function mapping from row to iterator of case classes.	`explode(lit(Array(2,3,-1)), "murh")`	`Row(2)`, `Row(3)`, `Row(-1)`

Beyond simply filtering out data, you can also produce a DataFrame with new columns or updated values in old columns. Spark uses the same expression syntax we discussed for filter, except instead of having to include a condition (like testing for equality), the results are used as values in the new DataFrame. To see how you can use `select` on complex and regular data types, Example 3-15 uses the Spark SQL `explode` function to turn an input DataFrame of `PandaPlaces` into a DataFrame of

just `PandaInfo` as well as computing the "squishness" to "hardness" ratio of each panda.

Example 3-15. Spark SQL select and explode operators

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

When you construct a sequence of operations, the generated column names can quickly become unwieldy, so the `as` or `alias` operator are useful to specify the resulting column name.

While all of these operations are quite powerful, sometimes the logic you wish to express is more easily encoded with `if/else` semantics. A simple example of this is encoding the different types of panda as a numeric value [5]. The `when` and `otherwise` functions can be chained together to create the same effect.

Example 3-16. If/Else in Spark SQL

```
/**
  * Encodes pandaType to Integer values instead of String values.
  *
  * @param pandaInfo the input DataFrame
  * @return Returns a DataFrame of pandaId and integer value for pandaType.
  */
def encodePandaType(pandaInfo: DataFrame): DataFrame = {
  pandaInfo.select(pandaInfo("id"),
    (when(pandaInfo("pt") === "giant", 0).
    when(pandaInfo("pt") === "red", 1).
    otherwise(2)).as("encodedType")
  )
}
```

5 StringIndexer in the ML pipeline is designed for string index encoding.

Specialized DataFrame Transformations for Missing & Noisy Data

Spark SQL also provides special tools for handling missing, null, and invalid data. By using `isNan` or `isNull` along with filters, you can create conditions for the rows you want to keep. For example, if you have a number of different columns, perhaps with different levels of precision (some of which may be null), you can use `coalesce(c1, c2, ..)` to return the first non-null column. Similarly, for numeric data, `nanvl` returns the first non-NaN value (e.g. nanvl(0/0, sqrt(-2), 3) results in 3). To simplify working with missing data, the `na` function on DataFrame gives us access to some common routines for handling missing data in DataFrameNaFunctions.

Beyond Row-by-Row Transformations

Sometimes applying a row-by-row decision, as you can with `filter`, isn't enough. Spark SQL also allows us to select the unique rows by calling `dropDuplicates`, but as with the similar operation on RDDs (`distinct`), this can require a shuffle, so is often much slower than filter. Unlike with RDDs, `dropDuplicates` can optionally drop rows based on only a subset of the columns, such as an id field.

Example 3-17. Drop duplicate panda ids

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

This leads nicely into our next section on aggregates and groupBy since often the most expensive component of each is the shuffle.

Aggregates and `groupBy`

Spark SQL has many powerful aggregates, and thanks to its optimizer it can be easy to combine many aggregates into one single action/query. Like with Pandas' DataFrames, `groupBy` returns a special `GroupedData` object on which we can ask for certain aggregations to be performed. Aggregations on Datasets behave differently returning a `GroupedDataset`, as discussed in "Grouped Operations on Datasets" on page 72. `min`, `max`, `avg`, and `sum` are all implemented as convenience functions directly on `GroupedData`, and more can be specified by providing the expressions to `agg`. Once you specify the aggregates you want to compute, you can get the results back as a DataFrame.

If you're used to RDDs you might be concerned by `groupBy`, but it is now a safe operation on DataFrames thanks to the Spark SQL optimizer, which automatically pipelines our reductions avoiding giant shuffles and mega records.

Example 3-18. Compute the max panda size by zip code

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

While this computes the max on a per-key basis, these aggregates can also be applied over the entire Dataframe or all numeric columns in a Dataframe. This is often useful when trying to collect some summary statistics for the data with which you are working. In fact there is a built-in `describe` transformation which does just that, although it can also be limited to certain columns.

Example 3-19. Compute some common summary stats, including count, mean, stddev, and more, on the entire DataFrame

```
pandas.describe()
```

The behaviour of groupBy has changed between Spark versions. Prior to Spark 1.3 the values of the grouping columns are discarded by default, while post 1.3 they are retained. The configuration parameter, `spark.sql.retainGroupColumns`, can be set to false to force the earlier functionality.

For computing multiple different aggregations, or more complex aggregations, you should use the `agg` api on the `GroupedData` instead directly calling `count`, `mean`, or similar convenience functions. For the `agg` API, you either supply a list of aggregate expressions, a string representing the aggregates, or a map of column names to aggregate function name. Once we've called `agg` with the requested aggregates, we get back a regular DataFrame with the aggregated results. As with regular functions, they are listed in org.apache.spark.sql.functions scaladoc. Table 3-8 lists some common and useful aggregates. For our example results in these tables we will consider a DataFrame with the schema of name field (as a string) and age (as an integer), both nullable with values (`{"ikea", null}, {"tube", 6}, {"real", 30}`).

Example 3-20. Example aggregates using the `agg` API

```
def minMeanSizePerZip(pandas: DataFrame): DataFrame = {
  // Compute the min and mean
  pandas.groupBy(pandas("zip")).agg(min(pandas("pandaSize")), mean(pandas("pandaSize")))
}
```

Computing multiple aggregates with Spark SQL can be much simpler than doing the same tasks with the RDD API.

Table 3-8. Spark SQL aggregate functions for use with agg API

Function name	Purpose	Storage requirement	Input Types	Example usage	Example Result
`approxCountDistinct`	Count approximate distinct values in column [a]	Configurable through rsd (which controls error rate)	All	`df.agg(approxCountDistinct(df("age"), 0.001))`	2
`avg`	Average	Constant	Numeric	`df.agg(avg(df("age")))`	{18}
`count`	Count number of items (excluding nulls). Special case of "*" counts number of rows	Constant	All	`df.agg(count(df("age")))`	2
`countDistinct`	Count distinct values in column	O(distinct elems)	All	`df.agg(countDistinct(df("age")))`	2
`first`	Return the first element [b]	Constant	All	`df.agg(first(df("age")))`	6
`last`	Return the last element	Constant	All	`df.agg(last(df("age")))`	30
`stddev`	Sample standard deviation [c]	Constant	Numeric	`df.agg(stddev(df("age")))`	16.97…
`stddev_pop`	Population standard deviation [c]	Constant	Numeric	`df.agg(stddev_pop(df("age")))`	12.0
`sum`	Sum of the values	Constant	Numeric	`df.agg(sum(df("age")))`	36

Function name	Purpose	Storage requirement	Input Types	Example usage	Example Result
`sumDistinct`	Sum of the distinct values	Constant	Numeric	`df.agg(sumDistinct(df("age")))`	36
`min`	Select the minimum value	Constant	Sortable data	`df.agg(min(df("age")))`	{5}
`max`	Select the maximum value	Constant	Sortable data	`df.agg(max(df("age")))`	{30}
`mean`	Select the maximum value	Constant	Sortable data	`df.agg(max(df("age")))`	{30}

[a] Implemented with HyperLogLog: *https://en.wikipedia.org/wiki/HyperLogLog* .

[b] This was commonly used in early versions of Spark SQL where the grouping column was not preserved.

[c] Added in Spark 1.6.

In addition aggregates on `groupBy`, you can run the same aggregations on multi-dimensional cubes with `cube` and rollups with `rollup`.

If the built-in aggregation functions don't meet your needs, you can extend Spark SQL using UDFS as discussed in "Extending with User Defined Functions & Aggregate Functions (UDFs, UDAFs)" on page 72, although things can be more complicated for aggregate functions.

Windowing

Spark SQL 1.4.0 introduced windowing functions to allow us to more easily work with ranges or windows of rows. When creating a window you specify what columns the window is over, the order of the rows within each partition/group, and the size of the window (e.g. K rows before and J rows after OR range between values). Using this specification each input row is related to some set of rows, called a frame, that is used to compute the resulting aggregate. Window functions can be very useful for things like computing average speed with noisey data, relative sales, and more.

Example 3-21. Define a window on the +/-10 closest (by age) pandas in the same zip code

```
val windowSpec = Window
  .orderBy(pandas("age"))
  .partitionBy(pandas("zip"))
  .rowsBetween(start = -10, end = 10) // use rangeBetween for range instead
```

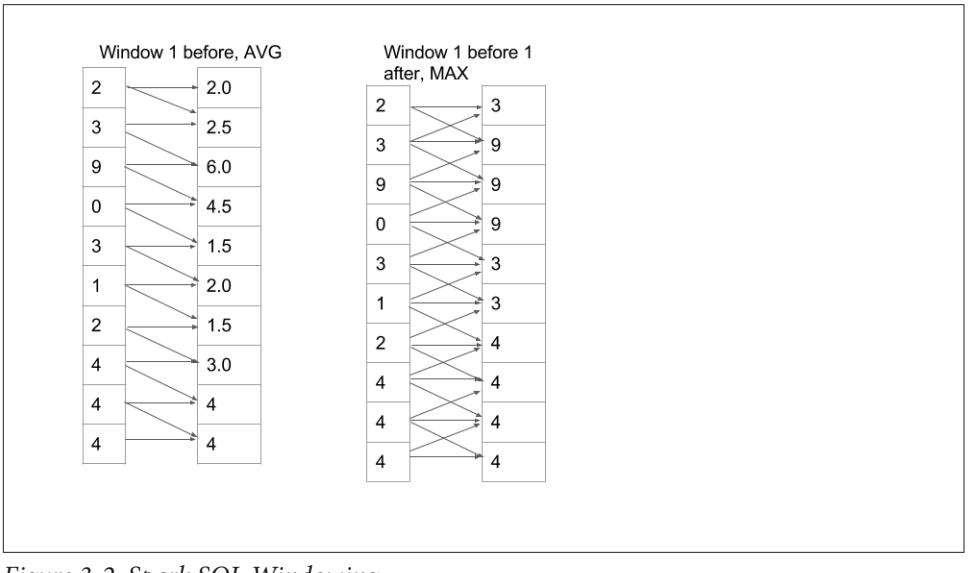


Figure 3-2. Spark SQL Windowing

Once you've defined a window specification you can compute a function over it. Spark's existing aggregate functions, covered in "Aggregates and groupBy" on page 50, can be computed an aggregate for the window. Window operations are very useful for things like Kalman filtering or many types of relative analysis.

Example 3-22. Compute difference from the average using the window of +/-10 closest (by age) pandas in the same zip code

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

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

As of this writing, Windowing functions require a HiveContext.

Sorting

Sorting supports multiple columns in ascending or descending order, with ascending as the default. Spark SQL has some extra benefits for sorting as some serialized data can be compared without deserialization.

Example 3-23. Sort by panda age and size in opposite orders

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

When limiting results, sorting is often used to only bring back the top or bottom K results. When limiting you specify the number of rows with `limit(numRows)` to restrict the number of rows in the DataFrame. Limits are also sometimes used for debuging without sorting to bring back a small result. If, instead of limiting the number of rows based on a sort order, you want to sample your data, ??? covers techniques for Spark SQL sampling as well.

Multi DataFrame Transformations

Beyond single DataFrame transformations you can perform operations that depend on multiple DataFrames. The ones that first pop into our heads are most likely the different types of joins, which are covered in Chapter 4, but beyond that you can also perform a number of set-like operations between DataFrames.

Set Like Operations

The DataFrame set-like operations allow us to perform many operations that are most commonly thought of as set operations. These operations behave a bit differently than traditional set operations since we don't have the restriction of unique elements. While you are likely already familiar with the results of set-like operations from regular Spark and Learning Spark, it's important to review the cost of these operations in Table 3-9.

Table 3-9. Set operations

Operation Name	Cost
`unionAll`	Low
`intersect`	Expensive
`except`	Expensive
`distinct`	Expensive

Plain Old SQL Queries and Interacting with Hive Data

Sometimes, it's to use regular SQL queries instead of building up our operations on DataFrames. If you are connected to a Hive Metastore we can directly write SQL queries against the Hive tables and get the results as a `DataFrame`. If you have a DataFrame you want to write SQL queries against, you can register it as a temporary table (or save it as a managed table if you intend to re-use it between jobs). Datasets can also be converted back to DataFrames and registered for querying against.

Example 3-24. Registering/saving tables

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

Querying tables is the same, regardless of whether it is a temporary table, existing Hive table, or newly-saved Spark table.

Example 3-25. Querying a table (permanent or temporary)

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

In addition to registering tables you can also write queries directly against a specific file path.

Example 3-26. Querying a raw file

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

Data Representation in DataFrames & Datasets

DataFrames are more than RDDs of `Row` objects; DataFrames and Datasets have a specialized representation and columnar cache format. The specialized representation is not only more space efficient, but also can be much faster to encode than even Kyro serialization. To be clear, like RDDs, DataFrames and Datasets are generally lazily evaluated and build up a lineage of their dependencies (except in DataFrames this is called a logical plan and contains more information).

Tungsten

Tungsten is a new Spark SQL component that provides more efficient Spark operations by working directly at the byte level. Looking back on, Figure 3-1, we can take a closer look at the space differences between the RDDs and DataFrames when cached in Figure 3-3. Tungsten includes specialized in-memory data structures tuned for the type of operations required by Spark, improved code generation, and a specialized wire protocol.

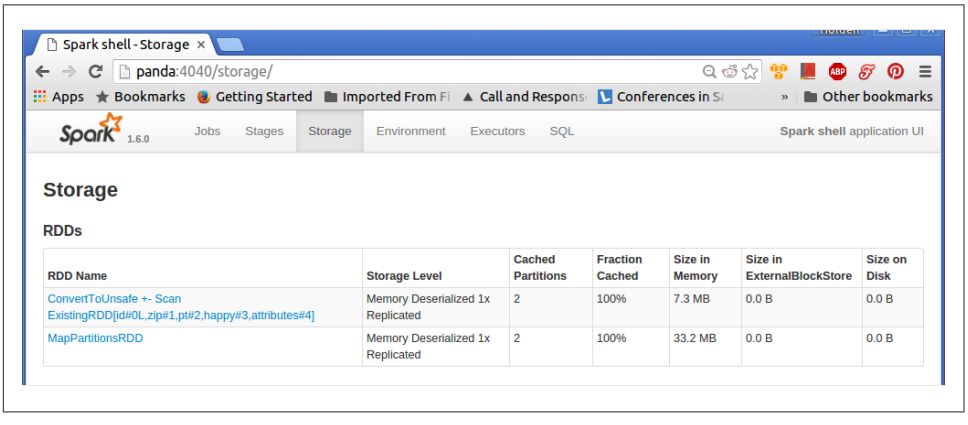


Figure 3-3. RDD versus Dataframe storage space for same data

For those coming from Hadoop, you can think of Tungsten data types as being `WritableComparable` types on steroids.

Tungsten's representation is substantially smaller than objects serialized using Java or even Kryo serializers. As Tungsten does not depend on Java objects, both on-heap and off-heap allocations are supported. Not only is the format more compact, serialization times can be substantially faster than with native serialization.

Since Tungsten no longer depends on working with Java objects, you can use either on-heap (in the JVM) or off-heap storage. If you use off-heap storage, it is important to leave enough room in your containers for the off-heap allocations - which you can get an approximate idea for from the web ui.

Tungsten's data structures are also created closely in mind with the kind of processing for which they are used. The classic example of this is with sorting, a common and

expensive operation. The on-wire representation is implemented so that sorting can be done without having to deserialize the data again.

In the future Tungsten may make it more feasible to use certain non-JVM libraries. For many simple operations the cost of using BLAS, or similar linear algebra packages, from the JVM is dominated by the cost of copying the data off-heap.

By avoiding the memory and GC overhead of regular Java objects, Tungsten is able to process larger data sets than the same hand-written aggregations. Tungsten became the default in Spark 1.5 and can be enabled in earlier versions by setting `spark.sql.tungsten.enabled` to true (or disabled in later versions by setting this to false). Even without Tungsten, Spark SQL uses a columnar storage format with Kryo serialization to minimize storage cost.

Data Loading and Saving Functions

Spark SQL has a different way of loading and saving data than core Spark. So as to be able to push down certain types of operations to the storage layer, Spark SQL has its own Data Source API. Data Sources are able to specify and control which type of operations should be pushed down to the data source. As developers, you don't need to worry too much about the internal activity going on here, unless the data sources you are looking for are not supported.

Data loading in Spark SQL is not quite as lazy as in regular Spark, but is still generally lazy. You can verify this by quickly trying to load from a data source that doesn't exist.

DataFrameWriter and DataFrameReader

The DataFrameWriter and the DataFrameReader cover writing and reading from external data sources. The `DataFrameWriter` is accessed by calling `write` on a `DataFrame` or `Dataset`. The `DataFrameReader` can be accessed through `read` on a `SQLContext`.

Spark SQL updated the load/save API in Spark 1.4, so you may see code still using the old-style API without the DataFrame reader or writer classes, but under the hood it is implemented as a wrapper around the new API.

Formats

When reading or writing you specify the format by calling `format(formatName)` on the `DataFrameWriter/DataFrameReader`. Format specific parameters, such as number of records to be sampled for JSON, are specified by either providing a map of options with `options` or setting option-by-option with `option` on the reader/writer.

The first party formats `json`, `jdbc`, `orc`, and `parquet` methods directly defined on the reader/writers taking the path or connection info. These methods are for convenience only and are wrappers around the more general methods we illustrate in this chapter.

JSON

Loading and writing JSON is supported directly in Spark SQL, and despite the lack of schema information in JSON, Spark SQL is able to infer a schema for us by sampling the records. Loading JSON data is more expensive than loading many data sources, since Spark needs to read some of the records to determine the schema information. If the schema between records varies widely (or the number of records is very small), you can increase the percentage of records read to determine the schema by setting `samplingRatio` to a higher value.

Example 3-27. Load JSON data, using all (100%) of records to determine the schema

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

Spark's schema inference can be a compelling reason to use Spark for processing JSON data, even if the data size could be handled on a single node.

Since our input may contain some invalid JSON records you may wish to filter out, you can also take in an RDD of strings. This allows us to load the input as a standard text file, filter out our invalid records, and then load the data into JSON. This is done by using the built-in `json` function on the DataFrameReader, which takes RDDs or paths. Methods for converting RDDs of regular objects are covered in "RDDs" on page 63.

Example 3-28. jsonRDD load

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

JDBC

The JDBC data source represent a natural Spark SQL data source, one which supports many of the same operations. Since different database vendors have slightly different JDBC implementations, you need to add the JAR for your JDBC data sources. Since SQL field types vary as well, Spark uses `JdbcDialects` with built-in dialects for DB2, Derby, MsSQL, Mysql, Oracle, and Postgres.[5]

While Spark supports many different JDBC sources, it does not ship with the JARs required to talk to all of these databases. If you are submitting your Spark job with `spark-submit` you can download the required JARs to the host you are launching and include them by specifying `--jars` or supply the maven coordinates to `--packages`. Since the Spark shell is also launched this way, the same syntax works and you can use it to include the MySQL JDBC JAR in Example 3-29.

Example 3-29. Include MySQL JDBC JAR

In earlier versions of Spark `--jars` does not include the JAR in the driver's class path. If this is the case for your cluster you must also specify the same JAR to `--driver-class-path`.

JdbcDialects allow Spark to correctly map the JDBC types to the corresponding Spark SQL types. If there isn't a JdbcDialect for your database vendor, the default dialect will be used which will likely work for many of the types. The dialect is automatically chosen based on the JDBC URL used.

If you find yourself needing to customize the JdbcDialect for your database vendor, you can look for a package or spark-packages or extend the `JdbcDialect` class and register your own dialect.

As with the other built-in data sources, there exists a convenience wrapper for specifying the properties required to load JDBC data. The convenience wrapper JDBC accepts the URL, table, and a `java.util.Properties` object for connection properties (such as authentication information). The properties object is merged with the properties that are set on the reader/writer itself. While the properties object is required, an empty properties object can be provided and properties instead specified on the reader/writer.

5 Some types may not be correctly implemented for all databases.

Example 3-30. Create a DataFrame from a JDBC data source

```
sqlContext.read.jdbc("jdbc:dialect:serverName;user=user;password=pass",
  "table", new Properties)
sqlContext.read.format("jdbc")
  .option("url", "jdbc:dialect:serverName")
  .option("dbtable", "table").load()
```

The API for saving a DataFrame is very similar to the API used for loading. The `save()` function needs no path since the information is already specified, just as with loading.

Example 3-31. Write a DataFrame to a JDBC data source

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

In addition to reading and writing JDBC data sources, Spark SQL can also run its own JDBC server, which is covered later on in this chapter.

Parquet

Apache Parquet files are a common format directly supported in Spark SQL, and they are incredibly space efficient and popular. Apache Parquet's popularity comes from a number of features, including the ability to easily split across multiple files, compression, nested types, and many others discussed in the Parquet documentation. Since Parquet is such a popular format, there are some additional options available in Spark for the reading and writing of Parquet files. Unlike third party data sources, these options are mostly configured on the SQLContext, although some can be configured on either the `SQLContext` or `DataFrameReader/Writer`.

Table 3-10. Parquet Data Source Options

SQLConf	DataFrameReader/ Writer option	Default	Purpose
spark.sql.parquet.mergeSchema	mergeSchema	False	Control if schema should be merged between partitions when reading. Can be expensive, so disabled by default in 1.5.0.

SQLConf	DataFrameReader/ Writer option	Default	Purpose
spark.sql.parquet.binaryAsString	N/A	False	Treat binary data as strings. Old versions of Spark wrote strings as binary data.
spark.sql.parquet.cacheMetadata	N/A	True	Cache parquet metadata, normally safe unless underlying data is being modified by another process.
spark.sql.parquet.compression.codec	N/A	Gzip	Specify the compression codec for use with parquet data. Valid options are uncompressed, snappy, gzip, or lzo.
spark.sql.parquet.filterPushdown	N/A	True	Push down filters to parquet (when possible) [a].
spark.sql.parquet.writeLegacyFormat	N/A	False	Write in parquet metadata in the legacy format.
spark.sql.parquet.output.committer.class	N/A	`org.apache.parquet.hadoop.ParquetOutputCommitter`	Output committer used by parquet. If writing to S3 you may wish to try `org.apache.spark.sql.parquet.DirectParquetOutputCommitter`.

[a] Pushdown means evaluate at the storage, so with parquet this can often mean skipping reading unnecessary rows or files.

Example 3-32. Read Parquet file written by an old version of Spark

```
def loadParquet(path: String): DataFrame = {
  // Configure Spark to read binary data as string, note: must be configured on SQLContext
  sqlContext.setConf("spark.sql.parquet.binaryAsString", "true")
  // Load parquet data using merge schema (configured through option)
  sqlContext.read
    .option("mergeSchema", "true")
    .format("parquet")
    .load(path)
}
```

Example 3-33. Write Parquet file with default options

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

Hive Tables

Interacting with Hive tables adds another option beyond the other formats. As covered in "Plain Old SQL Queries and Interacting with Hive Data" on page 56, one option for bringing in data from a Hive table is writing a SQL query against it and having the result as a DataFrame. The DataFrame's reader and writer interfaces can also be used with Hive tables as with the rest of the data sources.

Example 3-34. Load a Hive table

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

When loading a Hive table Spark SQL will convert the metadata and cache the result, if the underlying metadata has changed you can use `sqlContext.refreshTable("tablename")` to update the metadata, or the caching can be disabled by setting `spark.sql.parquet.cacheMetadata` to `false`.

Example 3-35. Write managed table

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

Unless specific conditions are met, the result saved to a Hive managed table will be saved in a Spark specific format that other tools may not be able to understand.

RDDs

Spark SQL DataFrames can easily be converted to RDDs of `Row` objects, and can also be created from RDDs of `Row` objects as well as JavaBeans, Scala case classes, and tuples. For RDDs of strings in JSON format, you can use the methods discussed in "JSON" on page 59. Datasets of type T can also easily be converted to RDDs of Type T, which can provide a useful bridge for DataFrames to RDDs of concrete case classes instead of `Row` objects. RDDs are a special case data source, since when going to/from RDDs, the data remains inside of Spark without writing out to or reading from an external system.

Converting a DataFrame to an RDD is a transformation (not an action); however, converting an RDD to a DataFrame or Dataset may involve computing (or sampling some of) the input RDD.

Creating a DataFrame from an RDD is not free in the general case. The data must be converted into Spark SQL's internal format.

When you create a DataFrame from an RDD Spark SQL needs to add schema information. If you are creating the DataFrame from an RDD of case classes or plain old Java objects (POJOs), Spark SQL is able to use reflection to automatically determine the schema. You can also manually specify the schema for our data using the structure discussed in "Basics of Schemas" on page 41. This can be especially useful if some of our fields are not nullable. You must specify the schema yourself if Spark SQL is unable to determine the schema through reflection, such as an RDD of Row objects (perhaps from calling `.rdd` on a DataFrame to use a functional transformation).

Example 3-36. Creating DataFrames from RDDs

```
def createFromCaseClassRDD(input: RDD[PandaPlace]) = {
  // Create DataFrame explicitly using sqlContext and schema inference
  val df1 = sqlContext.createDataFrame(input)

  // Create DataFrame using sqlContext implicits and schema inference
  val df2 = input.toDF()

  // Create a Row RDD from our RDD of case classes
  val rowRDD = input.map(pm => Row(pm.name,
    pm.pandas.map(pi => Row(pi.id, pi.zip, pi.happy, pi.attributes))))

  // Create DataFrame explicitly with specified schema
  val schema = StructType(List(StructField("name", StringType, true),
    StructField("pandas", ArrayType(StructType(List(
      StructField("id", LongType, true),
      StructField("zip", StringType, true),
      StructField("happy", BooleanType, true),
      StructField("attributes", ArrayType(FloatType), true)))))))

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

Case classes or JavaBeans defined inside another class can sometimes cause problems. If your RDD conversion is failing, make sure the case class being used isn't defined inside another class.

Converting a DataFrame to an RDD is incredibly simple; however, you get an RDD of `Row` objects. Since a row can contain anything, you need to specify the type (or cast the result) as you fetch the values for each column in the row. With Datasets you can directly get back an RDD templated on the same type, which can make the conversion back to a useful RDD much simpler.

While Scala has many implicit conversions for different numeric types, these do not generally apply in Spark SQL, instead we use explicit casting.

Example 3-37.

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

If you know that the schema of your DataFrame matches that of another, you can use the existing schema when constructing your new DataFrame. One common place where this occurs is when an input DataFrame has been converted to an RDD for functional filter and then back.

Local Collections

Much like with RDDs, you can also create DataFrames from local collections and bring them back as local collections. The same memory requirements apply; namely, the entire contents of the DataFrame will be in memory in the driver program. As such, distributing local collections is normally limited to unit tests, or joining small data sets with larger distributed datasets.

Example 3-38. Creating from a local collection

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

The LocalRelation's API we used here allows us to specify a schema in the same manner as when we are converting an RDD to a DataFrame.

In pre-1.6 versions of PySpark, schema inference only looked at the first record.

Collecting data back as a local collection is more common and often done post aggregations or filtering on the data. For example, with ML pipelines collecting the coefficents or in our Goldlilocks example collecting the quantiles to the driver. For larger data sets, saving to an external storage system (such as a database or HDFS) is recommended.

Just as with RDDs do not collect large DataFrames back to the driver. For Python users, it is important to remember that `toPandas()` collects the data locally.

Example 3-39. Collecting the Result locally

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

Additional Formats

As with core Spark, the data formats that ship directly with Spark only begin to scratch the surface of the types of systems with which you can interact. Some vendors publish their own implementations, and many are published on Spark Packages. As of this writing there are over twenty formats listed on the Data Source's page with the most popular being Avro, Redshift, CSV [5], and a unified wrapper around 6+ databases called deep-spark.

Spark packages can be included in our application in a few different ways. During the exploration phase (e.g. using the shell) you can include them by specifying `--packages` on the command line, as in Example 3-40. The same approach can be used when submitting our application with `spark-submit`, but this only includes the pack-

5 There is also a proposal to include csv directly in Spark since it is such a popular format.

age at run time, not at compile time. For including at compile time you can add the maven coordinates to our builds, or, if building with sbt, the sbt-spark-package plugin simplifies package dependencies with `spDependencies`.

Example 3-40. Starting Spark shell with CSV support

```
./bin/spark-shell --packages com.databricks:spark-csv_2.10:1.3.0
```

Example 3-41. Include spark-csv as an sbt dependency

```
  "com.databricks" % "spark-csv_2.10" % "1.3.0",
```

Once you've included the package with our Spark job you need to specify the format, as you did with the Spark provided ones. The name should be mentioned in the packages documentation. For spark-csv you would specify a format string of `com.databricks.spark.csv`.

There are a few options if the data format you are looking for isn't directly supported in either Spark or one of the libraries. Since many formats are available as Hadoop input formats, you can try to load our data as a Hadoop input format and convert the resulting RDD as discussed in "RDDs" on page 63. This approach is relatively simple, but means Spark SQL is unable to push down operations to our data store [5].

For a deeper integration you can implement our data source using the Data Source API . Depending on which operations you wish to support operator push-down for, your base relation you will need to implement additional traits from the `org.apache.spark.sql.sources` package. The details of implementing a new Spark SQL data source are beyond the scope of this book, but if you are interested the scaladoc for `org.apache.spark.sql.sources` and spark-csv's CsvRelation can be a good ways to get started.

Save Modes

In core Spark, saving RDDs always requires that the target directory does not exist, which can make appending to existing tables challenging. With Spark SQL, you can specify the desired behavior when writing out to a path that may already have data. The default behaviour is `SaveMode.ErrorIfExists`; matching RDDs behaviour, Spark will throw an exception if the target already exists. The different save modes and their behaviours are listed in Table 3-11.

5 For example, only reading the required partitions when a filter matches one of our partitioning schemes

Table 3-11. Save modes

Save Mode	Behaviour
ErrorIfExists	Throws an exception if the target already exists. If target doesn't exist write the data out.
Append	If target already exists, append the data to it. If the data doesn't exist write the data out.
Overwrite	If the target already exists, delete the target. Write the data out.
Ignore	If the target already exists, silently skip writing out. Otherwise write out the data.

Example 3-42. Specify save mode of append

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

Partitions (Discovery and Writing)

Partition data is an important part of Spark SQL since it powers one of the key optimizations to allow reading only the required data, discussed more in "Logical and Physical Plans" on page 75. If you know how our downstream consumers may access our data (e.g. reading data based on zip code), when you write your data it is beneficial to use that information to partition our output. When reading the data, it's useful to understand how partition discovery functions, so you can have a better understanding of whether our filter can be pushed down.

Filter push down can make a huge difference when working with large data sets by allowing Spark to only access the subset of data required for your computation instead of doing effectively a full table scan.

When reading partitioned data, you point Spark to the root path of your data, and it will automatically discover the different partitions. Not all data types can be used as partition keys; currently only strings and numeric data are the supported types.

If our data is all in a single DataFrame, the `DataFrameWriter` API makes it easy to specify the partition information while you are writing the data out. The `partitionBy` function takes a list of columns to partition the output on. You can also manually save our separate DataFrames (say if you are writing from different jobs) with individual `save` calls.

Example 3-43. Save partitioned by zip code

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

In addition to splitting the data by a partition key, it can be useful to make sure the resulting file sizes are reasonable, especially if the results will be used downstream by another Spark job. See ??? for general guidance.

Datasets

Datasets are an exciting extension of the DataFrame API which provide additional compile time type checking. Datasets can be used when our data can be encoded for Spark SQL and you know the type information at compile time. The Dataset API is a strongly typed collection with a mixture of relational (DataFrame) and functional (RDD) transformations. Like DataFrames, Datasets are represented by a logical plan the Catalyst optimizer can work with and when cached the data is stored in Spark SQL's internal encoding format.

The Dataset API is new in Spark 1.6 and will change in future versions. Users of the Dataset API are advised to treat it as a "preview." Up-to-date documentation on the Dataset API can be found in the scaladoc.

Interoperability with RDDs, DataFrames, and Local Collections

Datasets can be easily converted to/from DataFrames and RDDs, but in the initial version they do not directly extend either. Converting to/from RDDs involves encoding/decoding the data into a different form. Converting to/from DataFrames is almost "free" in that the underlying data does not need to be changed, only extra compile time type information is added/removed.

It is intended that future versions of DataFrames may extend `Dataset[Row]` to allow for easier interoperability.

To convert a DataFrame to a Dataset you can use the `as[ElementType]` function on the DataFrame to get a `Dataset[ElementType]` back as shown in Example 3-44. The `ElementType` must be a case class, or similar such as tuple, consisting of types Spark SQL can represent (see "Basics of Schemas" on page 41). To create `Datasets` from local collections, `createDataSet(...)` on the SQLContext and the `toDS()` implicit

are provided on Seqs in the same manner as `createDataFrame(...)` and `toDF()`. For converting from RDD to Dataset you can first convert from RDD to DataFrame and then convert it to a Dataset.

For loading data into a Dataset, unless a special API is provided by your data source, you can first load your data into a DataFrame and then convert it to a Dataset. Since the conversion to the Dataset simply adds information you do not have the problem of eagerly evaluating, and future filters and similar operations can still be pushed down to the data store.

Example 3-44. Create a Dataset from a DataFrame

Converting from a Dataset back to an RDD or DataFrame can be done in similar ways as when converting DataFrames. The `toDF` simply copies the logical plan used in the Dataset into a DataFrame - so you don't need to do any schema inference or conversion as you do when converting from RDDs. Converting a Dataset of type `T` to an RDD of type `T` can be done by calling `.rdd`, which unlike calling `toDF`, does involve converting the data from the internal SQL format to the regular types.

Example 3-45. Convert Dataset to DataFrame & RDD

```
/**
 * Illustrate converting a Dataset to an RDD
 */
def toRDD(ds: Dataset[RawPanda]): RDD[RawPanda] = {
  ds.rdd
}

/**
 * Illustrate converting a Dataset to a DataFrame
 */
def toDF(ds: Dataset[RawPanda]): DataFrame = {
  ds.toDF()
}
```

Compile Time Strong Typing

One of the reasons to use Datasets over traditional DataFrames is their compile time strong typing. DataFrames have runtime schema information but lack compile time information about the schema. This strong typing is especially useful when making libraries, because you can more clearly specify the requirements of your inputs and your return types.

Easier Functional (RDD "like") Transformations

One of the key advantages of the Dataset API is easier integration with custom Scala and Java code. Datasets expose `filter`, `map`, `mapPartitions`, and `flatMap` with similar function signatures as RDDs, with the notable requirement that our return ElementType also be understandable by Spark SQL (such as tuple or case class of types discussed in "Basics of Schemas" on page 41).

Example 3-46. Create a Dataset from a DataFrame

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

Beyond functional transformations, such as `map` and `filter`, you can also intermix relational and grouped operations.

Relational Transformations

Datasets introduce a typed version of `select` for relational style transformations. When specifying an expression for this you need to include the type information. You can add this information by calling `as[ReturnType]` on the expression/column.

Example 3-47. Simple relational select on Dataset

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

Some operations, such as select, have both typed and untyped implementations. If you supply a `Column` rather than a `TypedColumn` you will get a DataFrame back instead of a Dataset.

Multi-Dataset Relational Transformations

In addition to single Dataset transformations, there are also transformations for working with multiple Datasets. The standard set operations, namely `intersect`, `union`, and `subtract`, are all available with the same standard caveats as discussed in Table 3-9. Joining Datasets is also supported, but to make the type information easier to work with, the return structure is a bit different than traditional SQL joins.

Grouped Operations on Datasets

Similar to grouped operations on DataFrames, `groupBy` on Datasets return a `GroupedDataset`, on which you can specify your aggregate functions, along with a functional `mapGroups` API. As with the expression in "Relational Transformations" on page 71, you need to use typed expressions so the result can also be a Dataset. Taking our previous example of computing the maximum panda size by zip in Example 3-18, you would rewrite it to be as shown in Example 3-48.

The convenience functions found on `GroupedData` (e.g. `min`, `max`, etc.) are missing, so all of our aggregate expressions need to be specified through `agg`.

Example 3-48. Compute the max panda size per zip code typed

```
def maxPandaSizePerZip(ds: Dataset[RawPanda]): Dataset[(String, Double)] = {
  ds.groupBy($"zip").keyAs[String].agg(max("attributes(2)").as[Double])
}
```

Beyond applying typed SQL expressions to aggregated columns, you can also easily use arbitrary Scala code with `mapGroups` on grouped data as shown in Example 3-49. This can save us from having to write custom user defined aggregate functions (UDAFs) (discussed in "Extending with User Defined Functions & Aggregate Functions (UDFs, UDAFs)" on page 72). While custom UDAFs can be painful to write, they may be able to give better performance than `mapGroups` and can also be used on `DataFrames`.

Example 3-49. Compute the max panda size per zip code using map groups

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

Extending with User Defined Functions & Aggregate Functions (UDFs, UDAFs)

User defined functions and user defined aggregate functions provide you with ways to extend the DataFrame & SQL APIs with your own custom code while keeping the Catalyst optimizer. The Dataset API is another performant option for much of what you can do with UDFs and UDAFs. This is quite useful for performance, since otherwise you would need to convert the data to an RDD (and potentially back again) to

perform arbitrary functions, which is quite expensive. UDFs and UDAFs can also be accessed from inside of regular SQL expressions, making them accessible to analysts or others more comfortable with SQL.

When using UDFs or UDAFs written in non-JVM languages, it is important to note that you lose much of the performance benefit, as the data must still be transferred.

If most of your work is in Python but you want to access some UDFs without the performance penalty, you can write your UDFs in Scala and register them for use in Python (as done in Sparkling Pandas).[5]

Writing non-aggregate UDF for Spark SQL is incredibly simple: you simply write a regular function and register it using `sqlContext.udf().register`. If you are registering a Java or Python UDF you also need to specify our return type.

Example 3-50. String length UDF

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

Aggregate functions (or UDAFs) are somewhat trickier to write. Instead of writing a regular Scala function, you extend the `UserDefinedAggregateFunction` and implement a number of different functions, similar to the functions one might write for `aggregateByKey` on an RDD, except working with different data structures. While they can be complex to write, UDAFs can be quite performant compared with options like `mapGroups` on `Datasets` or even simply written `aggregateByKey` on `RDDs`. You can then either use the UDAF directly on columns or add it to the function registry as you did for the non-aggregate UDF.

Example 3-51. UDAF for computing the average

```
def setupUDAFs(sqlCtx: SQLContext) = {
  class Avg extends UserDefinedAggregateFunction {
    // Input type
    def inputSchema: org.apache.spark.sql.types.StructType =
      StructType(StructField("value", DoubleType) :: Nil)
```

5 As of this writing, the Sparkling Pandas project development is on-hold but early releases still contains some interesting examples of using JVM code from Python.

```
    def bufferSchema: StructType = StructType(
      StructField("count", LongType) ::
      StructField("sum", DoubleType) :: Nil
    )

    // Return type
    def dataType: DataType = DoubleType

    def deterministic: Boolean = true

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

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

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

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

This is a little more complicated than our regular UDF, so let's take a look at what the different parts do. You start by specifying what the input type is, then you specify the schema of the buffer you will use for storing the in-progress work. These schemas are specified in the same way as DataFrame and Dataset schemas, as discussed in "Basics of Schemas" on page 41.

From there the rest of the functions are implementing the same functions you use when writing `aggregateByKey` on an RDD, but instead of taking arbitrary Scala objects you work with `Row` and `MutableAggregationBuffer`. The final `evaluate` function takes the `Row` representing the aggregation data and returns the final result.

UDFs, UDAFs, and Datasets all provide ways to intermix arbitrary code with Spark SQL.

Query Optimizer

Catalyst is the Spark SQL query optimizer, which is used to take the query plan and transform it into an execution plan that Spark can run. Much as our transformations on RDDs build up a DAG, as you apply relational and functional transformations on DataFrames/Datasets, Spark SQL builds up a tree representing our query plan, called a logical plan. Spark is able to apply a number of optimizations on the logical plan and can also choose between multiple physical plans for the same logical plan using a cost-based model.

Logical and Physical Plans

The logical plan you construct through transformations on DataFrames/Datasets (or SQL queries) starts out as an unresolved logical plan. Much like a compiler, the Spark optimizer is multi-phased and before any optimizations can be performed, it needs to resolve the references and types of the expressions. This resolved plan is referred to as the logical plan, and Spark applies a number of simplifications directly on the logical plan, producing an optimized logical plan. These simplifications can be written using pattern matching on the tree, such as the rule for simplifying additions between two literals. The optimizer is not limited to pattern matching, and rules can also include arbitrary Scala code.

Once the logical plan has been optimized, Spark will produce a physical plan. The physical plan stage has both rule-based and cost-based optimizations to produce the optimal physical plan. One of the most important optimizations at this stage is predicate pushdown to the data source level.

Code Generation

As a final step, Spark may also apply code generation for the components. Code generation is done using Janino to compile Java code. Earlier versions used Scala's Quasi Quotes [5], but the overhead was too high to enable code generation for small datasets. In some TPCDS queries, code generation can result in >10x improvement in performance.

In some early versions of Spark for complex queries, code generation can cause failures. If you are on an old version of Spark and run into an unexpected failure it can be worth disabling codegen by setting `spark.sql.codegen` or `spark.sql.tungsten.enabled` to false (depending on version).

5 Scala Quasi Quotes are part of Scala's macro system.

JDBC/ODBC Server

Spark SQL provides a JDBC server to allow external tools, such as business intelligence, to work with data accessible in Spark and to share resources. Spark SQL's JDBC server requires that Spark be built with Hive support.

Since the server tends to be long lived and runs on a single context, it can also be a good way to share cached tables between multiple users.

Spark SQL's JDBC server is based on the HiveServer2 from Hive, and most corresponding connectors designed for HiveServer2 can be used directly with Spark SQL's JDBC server. Simba also offers specific drivers for Spark SQL.

The server can either be started from the command line or started using an existing HiveContext. The command line start and stop commands are `./sbin/start-thriftserver.sh` and `./sbin/stop-thriftserver.sh`. When starting from the command line, you can configure the different Spark SQL properties by specifying `--hiveconf property=value` on the command line. Many of the rest of the command line parameters match that of `spark-submit`. The default host and port is `localhost:10000` and can be configured with `hive.server2.thrift.port` and `hive.server2.thrift.bind.host`.

When starting the JDBC server using an existing `HiveContext`, you can simply update the config properties on the context instead of specifying command line parameters.

Example 3-52. Start JDBC server on a different port

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

Example 3-53. Start JDBC server on a different port in Scala

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

When starting the JDBC server on an existing `HiveContext` make sure to shutdown the JDBC server when exiting.

Conclusion

The considerations for using DataFrames/Datasets over RDDs are complex and changing with the rapid development of Spark SQL. One of the cases where Spark SQL can be difficult to use is when the number of partitions needed for different parts of our pipeline changes, or if you otherwise wish to control the partitioner. While RDDs lack the Catalyst optimizer and relational style queries, they are able to work with a wider variety of data types and provide more direct control over certain types of operations. DataFrames and Datasets also only work with a restricted subset of data types - but when our data is in one of these supported classes the performance improvements of using the Catalyst optimizer provide a compelling case for accepting those restrictions.

DataFrames can be used when you have primarily relational transformations, which can be extended with UDFS when necessary. Compared to RDDs, DataFrames benefit from the efficient storage format of Spark SQL, the Catalyst optimizer, and the ability to perform certain operations directly on the serialized data. One drawback to working with DataFrames is that they are not strongly typed at compile time, which can lead to errors with incorrect column access and other simple mistakes.

Datasets can be used when you want a mix of functional and relational transformations while benefiting from the optimizations for DataFrames and are, therefore, a great alternative to RDDs in many cases. As with RDDs, Datasets are parameterized on the type of data contained in them, which allows for strong compile time type checking but requires that you know our data type at compile time (although `Row` or other generic type can be used). The additional type safety of Datasets can be beneficial even for applications that do not need the specific functionality of DataFrames. One potential drawback is that the Dataset API is continuing to evolve, so updating to future versions of Spark may require code changes.

Pure RDDs work well for data that does not fit into the Catalyst optimizer. RDDs have an extensive and stable functional API, and upgrades to newer versions of Spark are unlikely to require substantial code changes. RDDs also make it easy to control partitioning, which can be very useful for many distributed algorithms. Some types of operations, such as multi-column aggregates, complex joins, and windowed operations, can be daunting to express with the RDD API. RDDs can work with any Java or Kryo serializable data, although the serialization is more often more expensive and less space efficient than the equivalent in DataFrames/Datasets.

Now that you have a good understanding of Spark SQL, it's time to continue on to joins, for both RDDs and Spark SQL.

CHAPTER 4

Joins (SQL & Core)

Joining data is an important part of many of our pipelines, and both Spark core and SQL support the same fundamental types of joins. While joins are very common and powerful, they warrant special performance consideration as they may require large network transfers or even create data sets beyond our capability to handle.[5]. In core Spark it can be more important to think about the ordering of operations, since the DAG optimizer, unlike the SQL optimizer isn't able to re-order or push down filters.

Core Spark Joins

In this section we will go over the RDD type joins. Joins in general are expensive since they require that corresponding keys from each RDD are located at the same partition so that they can be combined locally. If the RDDs do not have known partitioners, they will need to be shuffled so that both RDDs share a partitioner and data with the same keys lives in the same partitions, as shown in Figure 4-1. If they have the same partitioner, the data may be colocated, as in Figure 4-3, so as to avoid network transfer. Regardless of if the partitioners are the same, if one (or both) of the RDDs have a known partitioner only a narrow dependency is created, as in Figure 4-2. As with most key-value operations, the cost of the join increases with the number of keys and the distance the records have to travel in order to get to their correct partition.

5 As the saying goes, the cross product of big data and big data is an out of memory exception.

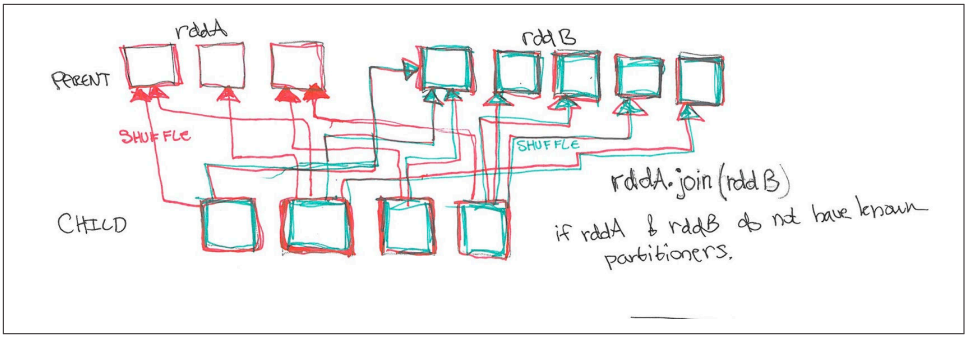


Figure 4-1. Shuffle join

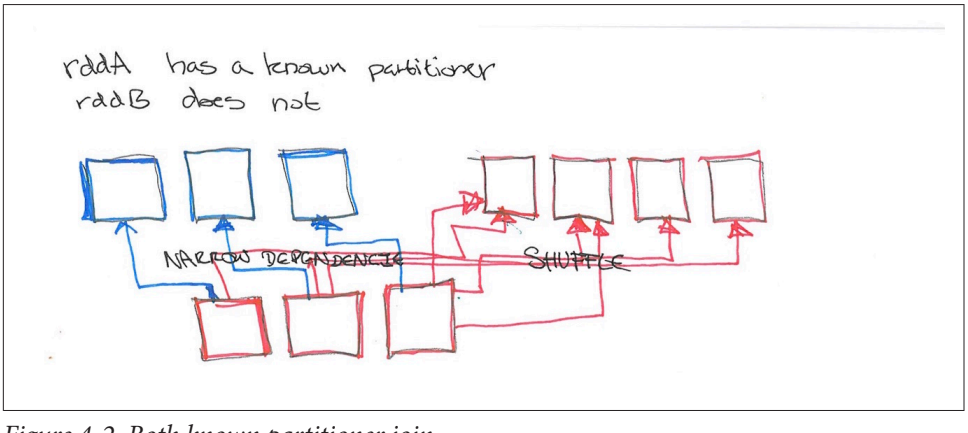


Figure 4-2. Both known partitioner join

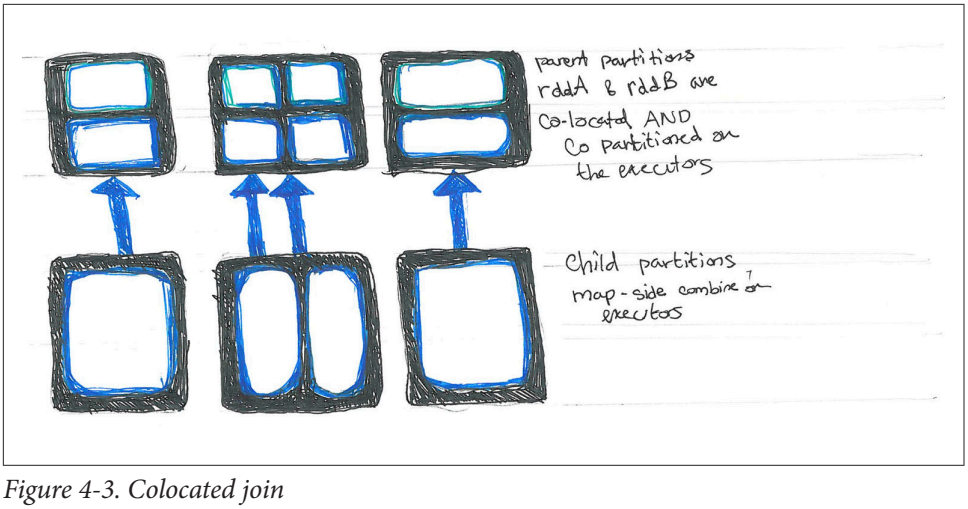


Figure 4-3. Colocated join

Two RDDs will be colocated if they have the same partitioner and were shuffled as part of the same action.

Core Spark joins are implemented using the `coGroup` function. We discuss `coGroup` in ???.

Choosing a Join Type

The default join operation in Spark includes only values for keys present in both RDDs, and in the case of multiple values per key, provides all permutations of the key/value pair. The best scenario for a standard join is when both RDDs contain the same set of distinct keys. With duplicate keys, the size of the data may expand dramatically causing performance issues, and if one key is not present in both RDDs you will loose that row of data. Here are a few guidelines:

1. When both RDDs have duplicate keys, the join can cause the size of the data to expand dramatically. It may be better to perform a `distinct` or `combineByKey` operation to reduce the key space or to use `cogroup` to handle duplicate keys instead of producing the full cross product. By using smart partitioning during the combine step, it is possible to prevent a second shuffle in the join (we will discuss this in detail later).
2. If keys are not present in both RDDs you risk losing your data unexpectedly. It can be safer to use an outer join, so that you are guaranteed to keep all the data in either the left or the right RDD, then filter the data after the join.
3. If one RDD has some easy-to-define subset of the keys, in the other you may be better off filtering or reducing before the join to avoid a big shuffle of data, which you will ultimately throw away anyway.

Join is one of the most expensive operations you will commonly use in Spark, so it is worth doing what you can to shrink your data before performing a join.

For example, suppose you have one RDD with some data in the form (Panda id, score) and another RDD with (Panda id, address), and you want to send each Panda some mail with her best score. You could join the RDDs on id and then compute the best score for each address. Like this:

Example 4-1. Basic RDD join

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

However, this is probably not as fast as first reducing the score data, so that the first dataset contains only one row for each Panda with her best score, and then joining that data with the address data.

Example 4-2. Pre-filter before join

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

}
```

If each Panda had 1000 different scores then the size of the shuffle we did in the first approach was 1000 times the size of the shuffle we did with this approach!

If we wanted to we could also perform a left outer join to keep all keys for processing even those missing in the right RDD by using `leftOuterJoin` in place of `join`. Spark also has `fullOuterJoin` and `rightOuterJoin` depending on which records we wish to keep. Any missing values are `None` and present values are `Some('x')`.

Example 4-3. Basic RDD left outer join

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

Choosing an Execution Plan

In order to join data, Spark needs the data that is to be joined (i.e. the data based on each key) to live on the same partition. The default implementation of join in Spark is a *shuffled hash join*. The shuffled hash join ensures that data on each partition will contain the same keys by partitioning the second dataset with the same default partitioner as the first, so that the keys with the same hash value from both datasets are in the same partition. While this approach always works, it can be more expensive than necessary because it requires a shuffle. The shuffle can be avoided if:

1. Both RDDs have a known partitioner.
2. One of the datasets is small enough to fit in memory, in which case we can do a broadcast hash join (we will explain what this is later).

Note that if the RDDs are colocated the network transfer can be avoided, along with the shuffle.

Speeding Up Joins by Assigning a Known Partitioner

If you have to do an operation before the join that requires a shuffle, such as `aggregateByKey` or `reduceByKey`, you can prevent the shuffle by adding a hash partitioner with the same number of partitions as an explicit argument to the first operation and persisting the RDD before the join. You could make the example in the previous section even faster, by using the partitioner for the address data as an argument for the `reduceByKey` step.

Example 4-4. Known partitioner join

```
def joinScoresWithAddress3( scoreRDD : RDD[(Long, Double)],
 addressRDD : RDD[(Long, String )]) : RDD[(Long, (Double, String))]= {
 //if addressRDD has a known partitioner we should use that,
 //otherwise it has a default hash parttioner, which we can reconstrut by getting the umber of
 // partitions.
 val addressDataPartitioner = addressRDD.partitioner match {
  case (Some(p)) => p
  case (None) => new HashPartitioner(addressRDD.partitions.length)
 }
 val bestScoreData = scoreRDD.reduceByKey(addressDataPartitioner, (x, y) => if(x > y) x else y)
 bestScoreData.join(addressRDD)
}
```

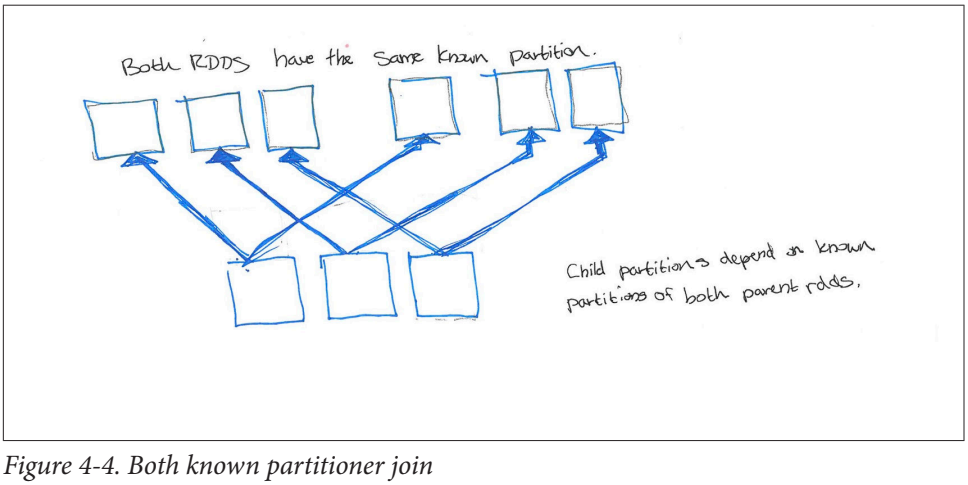


Figure 4-4. Both known partitioner join

Always persist after re-partitioning.

Speeding Up Joins Using a Broadcast Hash Join

A broadcast hash join pushes one of the RDDs (the smaller one) to each of the worker nodes. Then it does a map-side combine with each partition of the larger RDD. If one of your RDDs can fit in memory or can be made to fit in memory it is always beneficial to do a broadcast hash join, since it doesn't require a shuffle. Sometimes (but not always) Spark will be smart enough to configure the broadcast join itself. You can see what kind of join Spark is doing using the toDebugString() function.

Example 4-5. debugString

```
scoreRDD.join(addressRDD).toDebugString
```

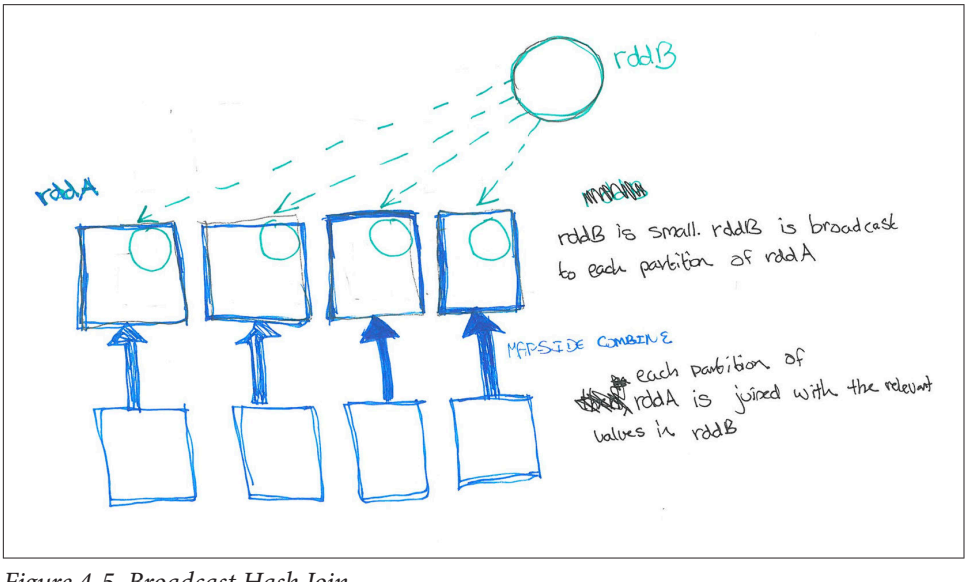


Figure 4-5. Broadcast Hash Join

Partial Manual Broadcast Hash Join

Sometimes not all of our smaller RDD will fit into memory, but some keys are so over-represented in the large data set, so you want to broadcast just the most common keys. This is especially useful if one key is so large that it can't fit on a single

partition. In this case you can use `countByKeyApprox` [5] on the large RDD to get an approximate idea of which keys would most benefit from a broadcast. You then filter the smaller RDD for only these keys, collecting the result locally in a HashMap. Using `sc.broadcast` you can broadcast the HashMap so that each worker only has one copy and manually perform the join against the HashMap. Using the same HashMap you can then filter our large RDD down to not include the large number of duplicate keys and perform our standard join, unioning it with the result of our manual join. This approach is quite convoluted but may allow you to handle highly skewed data you couldn't otherwise process.

Spark SQL Joins

Spark SQL supports the same basic join types as core Spark, but the optimizer is able to do more of the heavy lifting for you - although you also give up some of our control. For example, Spark SQL can sometimes push down or re-order operations to make our joins more efficient. On the other hand, you don't control the partitioner for DataFrames or Datasets, so you can't manually avoid shuffles as you did with core Spark joins.

DataFrame Joins

Joining data between DataFrames is one of the most common multi-DataFrame transformations. The standard SQL join types are supported and can be specified as the "joinType" when performing a join. As with joins between RDDs, joining with non-unique keys will result in the cross product (so if the left table has R1 and R2 with key1 and the right table has R3 and R5 with key1 you will get (R1, R3), (R1, R5), (R2, R3), (R2, R5)) in the output. While we explore Spark SQL joins we will use two example tables of pandas, Example 4-6 and Example 4-7.

While self joins are supported, you must alias the fields you are interested in to different names beforehand, so they can be accessed.

Example 4-6. Table of pandas and sizes

Name	Size
Happy	1.0

5 If the number of distinct keys is too high, you can also use `reduceByKey`, sort on the value, and take the top k.

Name	Size
Sad	0.9
Happy	1.5
Coffee	3.0

Example 4-7. Table of pandas and zip codes

Name	Zip
Happy	94110
Happy	94103
Coffee	10504
Tea	07012

Spark's supported join types are inner, left_outer (aliased as "outer"), right_outer and left_semi. With the exception of "left_semi" these join types all join the two tables, but they behave differently when handling rows that do not have keys in both tables.

The "inner" join is both the default and likely what you think of when you think of joining tables. It requires that the key be present in both tables, or the result is dropped as shown in Example 4-8 and inner join table.

Example 4-8. Simple inner join

```
// Inner join implicit
df1.join(df2, df1("name") === df2("name"))
// Inner join explicit
df1.join(df2, df1("name") === df2("name"), "inner")
```

Table 4-1. Inner join of df1, df2 on name

Name	Size	Name	Zip
Coffee	3.0	Coffee	10504
Happy	1.0	Happy	94110
Happy	1.5	Happy	94110
Happy	1.0	Happy	94103

Name	Size	Name	Zip
Happy	1.5	Happy	94103

Left outer joins will produce a table with all of the keys from the left table, and any rows without matching keys in the right table will have null values in the fields that would be populated by the right table. Right outer joins are the same, but with the requirements reversed.

Example 4-9. Left outer join

```
// Left outer join explicit
df1.join(df2, df1("name") === df2("name"), "left_outer")
```

Table 4-2. Left outer join df1, df2 on name

Name	Size	Name	Zip
Sad	0.9	null	null
Coffee	3.0	Coffee	10504
Happy	1.0	Happy	94110
Happy	1.5	Happy	94110
Happy	1.5	Happy	94103

Example 4-10. Right outer join

```
// Right outer join explicit
df1.join(df2, df1("name") === df2("name"), "right_outer")
```

Table 4-3. Right outer join df1, df2 on name

Name	Size	Name	Zip
Sad	0.9	null	null
Coffee	3.0	Coffee	10504
Happy	1.0	Happy	94110
Happy	1.5	Happy	94110
Happy	1.5	Happy	94103

Name	Size	Name	Zip
null	null	Tea	07012

Left semi joins are the only kind of join which only has values from the left table. A left semi join is the same as filtering the left table for only rows with keys present in the right table.

Example 4-11. Left semi join

```
// Left semi join explicit
df1.join(df2, df1("name") === df2("name"), "leftsemi")
```

Table 4-4. Left semi join

Name	Size
Coffee	3.0
Happy	1.0
Happy	1.5

Self Joins

Self joins are supported on DataFrames; but we end up with duplicated columns names. So you can access the results you need to alias the DataFrames to different names. Once you've aliased each DataFrame, in the result you can access the individual columns for each DataFrame with `dfName.colName`.

Example 4-12. Self join

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

Broadcast Hash Joins

In SparkSQL you can see the type of join being performed by calling `queryExecution.executedPlan`. As with core Spark, if one of the tables is much smaller than the other you may want a broadcast hash join. You can hint to Spark SQL that a given DF should be broadcast for join by calling `broadcast` on the DataFrame before joining it (e.g. `df1.join(broadcast(df2), "key")`). Spark also automatically uses the `spark.sql.conf.autoBroadcastJoinThreshold` to determine if a table should be broadcast.

Dataset Joins

Joining Datasets is done with `joinWith`, and this behaves similarly to a regular relational join, except the result is a tuple of the different record types as shown in Example 4-13. This is somewhat more awkward to work with after the join, but also does make self joins, as shown in Example 4-14, much easier, as you don't need to alias the columns first.

Example 4-13. Joining two Datasets

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

Example 4-14. Self join a Dataset

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

Using a self join and a `lit(true)`, you can produce the cartesian product of your Dataset, which can be useful but also illustrates how joins (especially self-joins) can easily result in unworkable data sizes.

As with DataFrames you can specify the type of join desired (e.g. inner, left_outer, right_outer, leftsemi), changing how records present only in one Dataset are handled. Missing records are represented by null values, so be careful.

Conclusion

Now that you have explored joins, it's time to focus on transformations and the performance considerations associated with them. For those interested in continuing learning more about Spark SQL, we will continue with Spark SQL tuning in ???, where we include more details on join-specific configurations like number of partitions and join thresholds.