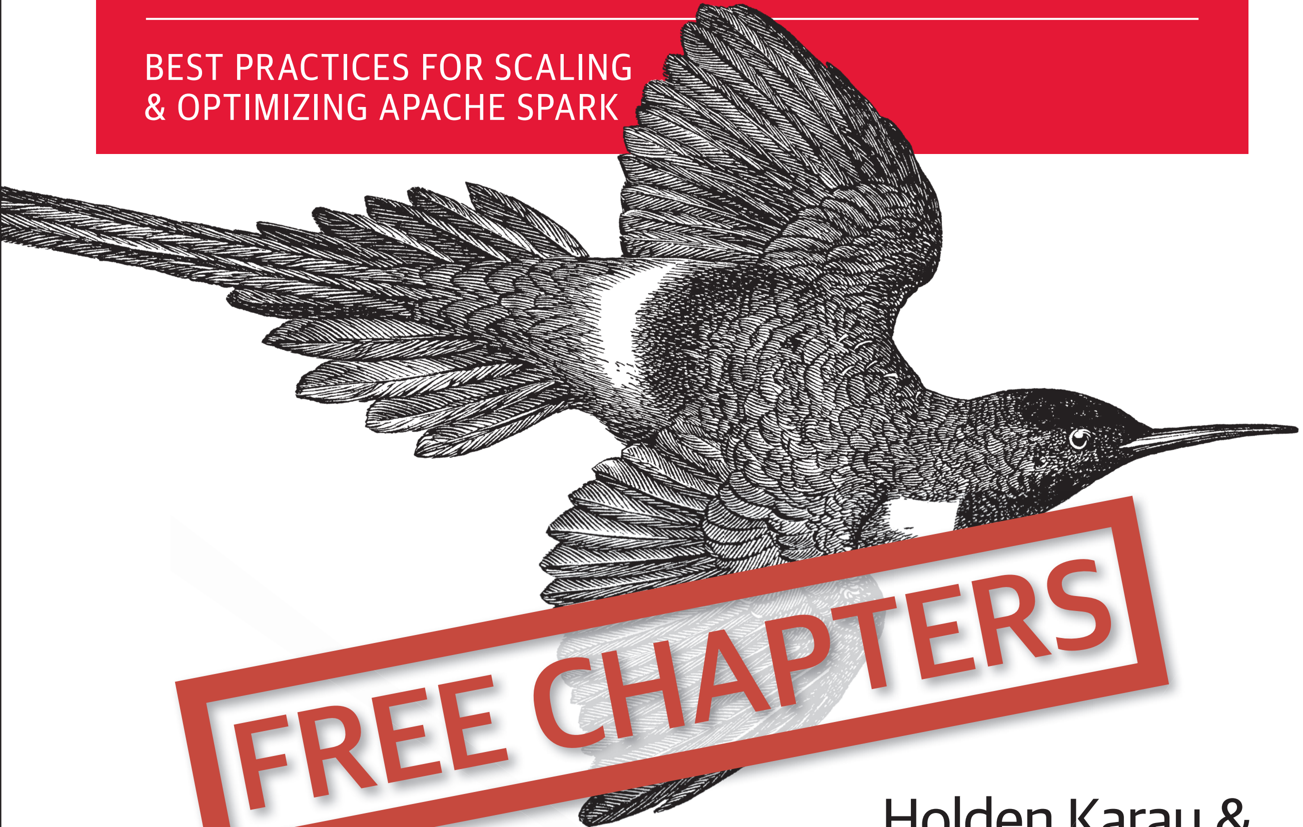
O'REILLY®

Compliments of MESOSPHERE

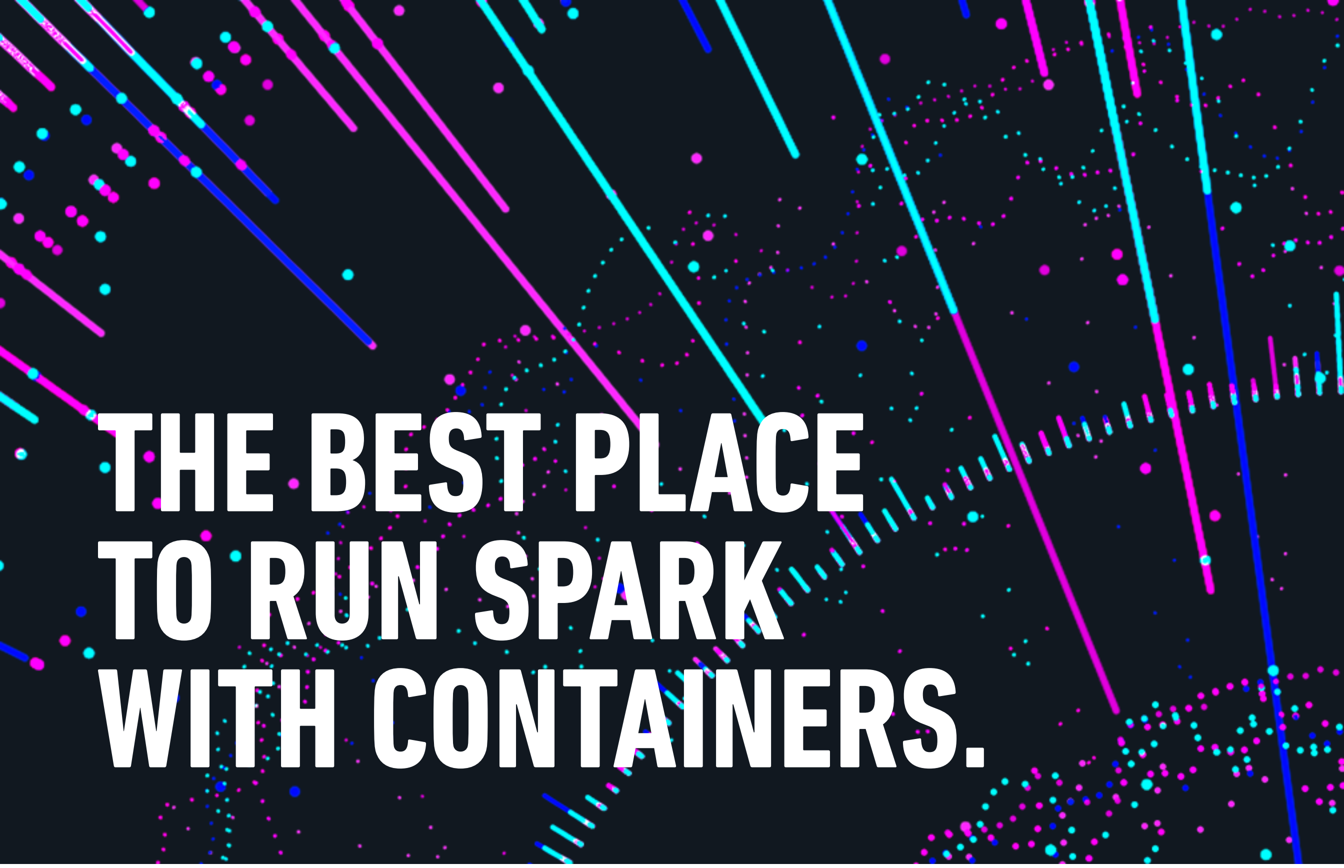
High Performance Spark

BEST PRACTICES FOR SCALING & OPTIMIZING APACHE SPARK



FREE CHAPTERS

Holden Karau & Rachel Warren



THE BEST PLACE TO RUN SPARK WITH CONTAINERS.

Mesosphere DC/OS makes it easy to build and elastically scale data-rich applications.

- 1-Click Install & Operation of Data Services
- Run on Any Cloud, Public or Private
- Secure & Proven in Production
- Dramatically Cuts Infrastructure Costs

LEARN MORE →

MESOSPHERE

High Performance Spark

Best Practices for Scaling and Optimizing Apache Spark

This Excerpt contains Chapters 1, 2, and Appendix A of the book *High Performance Spark*. The complete book is available at oreilly.com and through other retailers.

Holden Karau and Rachel Warren

Beijing · Boston · Farnham · Sebastopol · Tokyo

O'REILLY®

High Performance Spark
by Holden Karau and Rachel Warren

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://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Shannon Cutt	**Indexer:** Ellen Troutman-Zaig
Production Editor: Kristen Brown	**Interior Designer:** David Futato
Copyeditor: Kim Cofer	**Cover Designer:** Karen Montgomery
Proofreader: James Fraleigh	**Illustrator:** Rebecca Demarest

June 2017: First Edition

Revision History for the First Edition
2017-05-22: First Release

978-1-491-94320-5

[LSI]

Table of Contents

Foreword

Data is truly the new oil. The ability to easily access tremendous amounts of computing power has made data the new basis of competition. We've moved beyond traditional big data, where companies gained historical insights using batch analytics—in today's digital economy, businesses must learn to extract value from data and build modern applications that serve customers with personalized services, in real time, and at scale.

At Mesosphere, we are seeing two major technology trends that enable modern applications to handle users and data at scale: containers and data services. Microservice-based applications that are deployed in containers speed time to market and reduce infrastructure overhead, among other benefits. Data services capture, process, and store data being used by the containerized microservices.

A de facto architecture is emerging to build and operate fast data applications, with a set of leading technologies used within this architecture that are scalable, enable real-time processing, and open source. This set of technologies is often referred to as the "SMACK" stack:

Apache Kafka
: A distributed, highly available messaging system to ensure millions of events per second are captured through connected endpoints with no loss

Apache Spark
: A large-scale analytics engine that supports streaming, machine learning, SQL, and graph computation

Apache Cassandra
: A distributed database that is highly available and scalable

Akka
: A toolkit and runtime to simplify development of data-driven apps

Apache Mesos
: A cluster resource manager that serves as a highly available, scalable, and efficient platform for running data services and containerized microservices.

High Performance Spark was written for data engineers and data scientists who are looking to get the most out of Spark. The book lays out the key strategies to make Spark queries faster, able to handle larger datasets, and use fewer resources.

Mesosphere is proud to offer this excerpt, as we were founded with the goal of making cutting edge technologies such as Spark and the SMACK stack easy to use. In fact, Spark was created as the first proof-of-concept workload to run on Mesos. Mesos serves as an elastic and proven foundation for building and elastically scaling data-rich, modern applications. We created DC/OS to make Mesos simple to use, including the automation of data services.

We hope you enjoy the excerpt, and that you consider Mesosphere DC/OS to jump start your journey to building, deploying, and scaling a data-intensive application that helps your business.

— Tobi Knaup
Chief Technology Officer, Mesosphere

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).

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 1,000 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 that 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 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 toward the structure of the data and the requirements of the cluster,

1 From *http://spark.apache.org/*.

succeeds on the same system with terabytes of data. In the authors' experience writing production Spark code, we have seen the same tasks, run on the same clusters, run 100× 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 and is 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 our 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 just as quick as the alternatives that we will present. 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 that might not apply to the problems you are working with, but that might apply to a problem in the future and may help shape your understanding of Spark more 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, Wendell, and Zaharia as well as Paco Nathan's introduction 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 multitenancy are not covered. We are assuming that you already have a way to use Spark in your system, so we 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.

Spark Versions

Spark follows semantic versioning with the standard [MAJOR].[MINOR].[MAINTENANCE] with API stability for public nonexperimental nondeveloper APIs within minor and maintenance releases. 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[2]; so if you are using the stable API you generally should not need to recompile to run a job against a new version of Spark unless the major version has changed.

This book was created using the Spark 2.0.1 APIs, 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.

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 codebase 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 *Resilient Distributed Datasets* (RDD) class closely mimic those in the Scala collections API. RDD functions, such as `map`, `filter`, `flatMap`,

2 MiMa is the Migration Manager for Scala and tries to catch binary incompatibilities between releases.

`reduce`, and `fold`, have nearly identical specifications to their Scala equivalents.[3] Fundamentally Spark is a functional framework, relying heavily on concepts like immutability and lambda definition, so using the Spark API may be more intuitive with some knowledge of 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 inline 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 is only available in languages with existing REPLs (Scala, Python, and R).

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 toolkits. 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 reasons 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).[4]

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

4 Of course, in performance, every rule has its exception. `mapPartitions` in Spark 1.6 and earlier in Java suffers some severe performance restrictions that we discuss in Chapter 5.

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 email us or create an issue on the GitHub repo.

Spark SQL does much to minimize the performance difference when using a non-JVM language. Chapter 7 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. Spark 1.6 is built against Scala 2.10 and cross-compiled against Scala 2.11, and Spark 2.0 is built against Scala 2.11 and possibly cross-compiled against Scala 2.10 and may add 2.12 in the future. 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 massive open online courses (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 Lightbend (formerly 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 Chapter 7. 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 paradigms that are important to understanding how to efficiently utilize Spark.

CHAPTER 2

How Spark Works

This chapter introduces the overall design of Spark as well as its place in the big data ecosystem. Spark is often considered an alternative to Apache MapReduce, since Spark can also be used for distributed data processing with Hadoop.[1] As we will discuss in this chapter, Spark's design principles are quite different from those of MapReduce. Unlike Hadoop MapReduce, Spark does not need to be run in tandem with Apache Hadoop—although it often is. Spark has inherited parts of its API, design, and supported formats from other existing computational frameworks, particularly DryadLINQ.[2] However, Spark's internals, especially how it handles failures, differ from many traditional systems. Spark's ability to leverage lazy evaluation within memory computations makes it particularly unique. Spark's creators believe it to be the first high-level programming language for fast, distributed data processing.[3]

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 a thorough explanation of the Spark scheduler and execution engine. We will refer to

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 *Hadoop MapReduce*, packaged with the distributed filesystem, Apache Hadoop Distributed File System.

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

3 See the original Spark Paper and other Spark papers.

the concepts in this chapter throughout the text. Further, we hope this explanation will provide you with a more precise understanding of some of the terms you've heard tossed around by other Spark users and encounter in the Spark documentation.

How Spark Fits into the Big Data Ecosystem

Apache Spark is an open source framework that provides methods to process data in parallel that are generalizable; the same high-level Spark functions can be used to perform disparate data processing tasks on data of different sizes and structures. On its own, Spark is not a data storage solution; it performs computations on Spark JVMs (Java Virtual Machines) that last only for the duration of a Spark application. 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 (e.g., HDFS, Cassandra, or S3) and a cluster manager—the storage system to house the data processed with Spark, and the cluster manager to orchestrate the distribution of Spark applications across the cluster. Spark currently supports three kinds of cluster managers: Standalone Cluster Manager, Apache Mesos, and Hadoop YARN (see Figure 2-1). The Standalone Cluster Manager is included in Spark, but using the Standalone manager requires installing Spark on each node of the cluster.

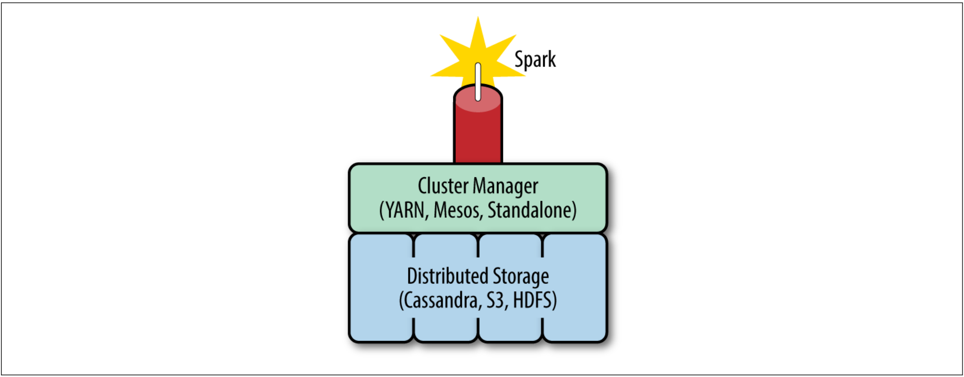


Figure 2-1. A diagram of the data processing ecosystem including Spark

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, Python, and R. 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 (functions that are applied to the entire dataset), such as `map`, `join`, and `reduce` to manipulate the distributed datasets, as well as I/O functionality to read and write data between the distributed storage system and the Spark JVMs.

While Spark also supports R, at present the RDD interface is not available in that language. We will cover tips for using Java, Python, R, and other languages in detail in Chapter 7.

In addition to Spark Core, the Spark ecosystem includes a number of other first-party components, including Spark SQL, Spark MLlib, Spark ML, Spark Streaming, and GraphX,[4] which provide more specific data processing functionality. Some of these components have the same generic performance considerations as the Core; MLlib, for example, is written almost entirely on the Spark API. However, some of them have unique considerations. Spark SQL, for example, has a different query optimizer than Spark Core.

Spark SQL is a component that can be used in tandem with Spark Core and has APIs in Scala, Java, Python, and R, and basic SQL queries. Spark SQL defines an interface for a semi-structured data type, called `DataFrames`, and as of Spark 1.6, a semi-structured, typed version of RDDs called called `Datasets`.[5] Spark SQL is a very important component for Spark performance, and much of what can be accomplished with Spark Core can be done by leveraging Spark SQL. We will cover Spark SQL in detail in Chapter 3 and compare the performance of joins in Spark SQL and Spark Core in Chapter 4.

Spark has two machine learning packages: ML and MLlib. MLlib is a package of machine learning and statistics algorithms written with Spark. Spark ML is still in the early stages, and has only existed since Spark 1.2. Spark ML provides a higher-level API than MLlib with the goal of allowing users to more easily create practical machine learning pipelines. Spark MLlib is primarily built on top of RDDs and uses functions from Spark Core, while ML is built on top of Spark SQL `DataFrames`.[6] Eventually the Spark community plans to move over to ML and deprecate MLlib. Spark ML and MLlib both have additional performance considerations from Spark Core and Spark SQL—we cover some of these in Chapter 9.

Spark Streaming uses the scheduling of the Spark Core for streaming analytics on minibatches of data. Spark Streaming has a number of unique considerations, such as

4 GraphX is not actively developed at this point, and will likely be replaced with GraphFrames or similar.

5 `Datasets` and `DataFrames` are unified in Spark 2.0. `Datasets` are `DataFrames` of "Row" objects that can be accessed by field number.

6 See the MLlib documentation.

the window sizes used for batches. We offer some tips for using Spark Streaming in Chapter 10.

GraphX is a graph processing framework built on top of Spark with an API for graph computations. GraphX is one of the least mature components of Spark, so we don't cover it in much detail. In future versions of Spark, typed graph functionality will be introduced on top of the Dataset API. We will provide a cursory glance at GraphX in Chapter 10.

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. Maybe by the time you're done, you will be ready to start contributing your own functions to MLlib and ML!

In addition to these first-party components, the community has written a number of libraries that provide additional functionality, such as for testing or parsing CSVs, 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 your `maven` or `sbt` project. We first use Spark packages to add support for CSV data in Chapter 3 and then in more detail in Chapter 10.

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 immutability allows Spark to be easy-to-use, fault-tolerant, scalable, and efficient.

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 an action is called. An action is a Spark operation that returns something other than an RDD, triggering evaluation of partitions and possibly returning some output to a non-Spark system (outside of the Spark executors); for example, bringing data back to the driver (with operations like `count` or `collect`) or writing data to an 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 result.

Not all transformations are 100% lazy. `sortByKey` needs to evaluate the RDD to determine the range of data, so it involves both a transformation and an action.

Performance and usability advantages of lazy evaluation

Lazy evaluation allows Spark to combine 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 a Spark program calls a `map` and a `filter` function on the same RDD. Spark can send the instructions for both the `map` and the `filter` to each executor. Then Spark can perform both the `map` and `filter` on each partition, which requires accessing the records only once, rather than sending two sets of instructions and accessing each partition twice. This theoretically reduces 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—that requires the developer to do the work to consolidate her mapping operations. Spark's clever lazy evaluation strategy lets us be lazy and express the same logic in far fewer lines of code: 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 that, given a dataset of documents, parses the text into words and then computes the count for each word. The Apache docs provide a word count example, which even in its simplest form comprises roughly fifty lines of code (excluding import statements) in Java. A comparable Spark implementation is roughly fifteen lines of code in Java and five in Scala, available on the Apache website. The example excludes the steps to read in the data mapping documents to words and counting the words. We have reproduced it in Example 2-1.

Example 2-1. Simple Scala word count example

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

A further benefit of the Spark implementation of word count is that it is easier to modify and improve. Suppose that we now want to modify this function to filter out some "stop words" and punctuation from each document before computing the word count. In MapReduce, 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 preceding Spark routine by simply putting a `filter` step before the `map` step that creates the key/value pairs. Example 2-2 shows how Spark's lazy evaluation will consolidate the `map` and `filter` steps for us.

Example 2-2. Word count example with stop words filtered

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

Lazy evaluation and fault tolerance

Spark is fault-tolerant, meaning Spark will not fail, lose data, or return inaccurate results in the event of a host machine or network failure. Spark's unique method of fault tolerance is achieved because each partition of the data contains the dependency information needed to recalculate the partition. Most distributed computing paradigms that allow users to work with mutable objects 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.

Lazy evaluation and debugging

Lazy evaluation has important consequences for debugging since it means that a Spark program will fail only at the point of action. For example, suppose that you were using the word count example, and afterwards were collecting the results to the driver. If the value you passed in for the stop words was null (maybe because it was the result of a Java program), the code would of course fail with a null pointer exception in the `contains` check. However, this failure would not appear until the program evaluated the collect step. Even the stack trace will show the failure as first occurring at the collect step, suggesting that the failure came from the collect statement. For this reason it is probably most efficient to develop in an environment that gives you access to complete debugging information.

Because of lazy evaluation, stack traces from failed Spark jobs (especially when embedded in larger systems) will often appear to fail consistently at the point of the action, even if the problem in the logic occurs in a transformation much earlier in the program.

In-Memory Persistence and Memory Management

Spark's performance advantage over MapReduce is greatest in use cases involving repeated computations. Much of this performance increase is due to Spark's use of in-memory persistence. 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 as deserialized data, in-memory as serialized data, and on disk. Each has different space and time advantages:

In memory as deserialized Java objects
: The most intuitive way to store objects in RDDs is as the original 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 stored as objects.

As serialized data

Using the standard 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. While Java serialization is more efficient than full objects, Kryo serialization (discussed in "Kryo" on page 42) can be even more space efficient.

On disk

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 sequences 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 Chapter 5. When persisting RDDs, the default implementation of RDDs evicts the least recently used partition (called LRU caching) if the space it takes is required to compute or to cache a new partition. However, you can change this behavior and control Spark's memory prioritization with the `persistencePriority()` function in the RDD class. See Chapter 5.

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 three ways: (1) by transforming an existing RDD; (2) from a `SparkContext`, which is the API's gateway to Spark for your application; and (3) converting a `DataFrame` or `Dataset` (created from the `SparkSession`[7]). The `SparkContext` represents the connection between a Spark cluster and one running Spark application. The `SparkContext` can be used to create an RDD from a local Scala object (using the `makeRDD` or `parallelize` methods) or by reading from stable storage

7 Prior to Spark 2.0, the `SparkSession` was called the `SQLContext`.

(text files, binary files, a Hadoop Context, or a Hadoop file). `DataFrames` and `Datasets` can be read using the Spark SQL equivalent to a `SparkContext`, the `SparkSession`.

Internally, Spark uses five main properties to represent an RDD. The three required properties are the list of partition objects that make up the RDD, 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). As an end user, you will rarely need these five properties and are more likely to use predefined RDD transformations. However, it is helpful to understand the properties and know how to access them for debugging and for a better conceptual understanding. 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 Versus Narrow Dependencies" on page 17.

`partitioner()`
: Returns a Scala option type of a `partitioner` object if the RDD has a function between `element` and `partition` 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 partition for each block of the file. We will discuss partitioning in detail in Chapter 6.

`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 `preferredLocations` is the Hadoop name of the node where that partition is stored.

Types of RDDs

The implementation of the Spark 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 function 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 implementations of the `RDD` class that define more specific behavior by overriding the core properties of the RDD. These include the `NewHadoopRDD` class discussed previously—which represents an RDD created from an HDFS filesystem—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 `SparkContext`, 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.

> Find out what type an RDD is by using the `toDebugString` function, which is defined on all RDDs. This will tell you what kind of RDD you have and provide a list of its parent RDDs.

We will discuss the different types of RDDs and RDD transformations in detail in Chapters 5 and 6.

Functions on RDDs: Transformations Versus Actions

There are two types of functions defined on RDDs: *actions* and *transformations*. Actions are functions that return something that is not an RDD, including a side effect, and transformations are functions 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 are what force evaluation of a Spark program. Persist calls also force evaluation, but usually do not mark the end of Spark job. 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`. Most 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 of tuple type by implicit conversion) and the `NewHadoopRDD` class, which is an implementation for RDDs that were created by reading from Hadoop. Some saving functions, like `saveAsTextFile` and `saveAsObjectFile`, are available on all RDDs, and they work by adding an implicit null key to each record (which is then ignored by the saving level). Functions that return nothing (*void* in Java, or `Unit` in Scala), such as `foreach`, are also actions: they force execution of a Spark job. `foreach` can be used to force evaluation of an RDD, but is also often used to write out to nonsupported formats (like web endpoints).

Most of the power of the Spark API is in its transformations. Spark transformations are coarse-grained transformations used to sort, reduce, group, sample, filter, and map distributed data. We will discuss transformations in detail in both Chapter 6, which deals exclusively with transformations on RDDs of key/value data, and Chapter 5.

Wide Versus 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 versus 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 19 of this chapter, but we will save the longer explanation of the performance considerations associated with them for Chapter 5.

Conceptually, narrow transformations are those in which each partition in the child RDD has simple, finite dependencies on partitions in the parent RDD. Dependencies are only narrow if they can be determined at design time, irrespective of the values of the records in the parent partitions, and if each parent has at most one child partition. Specifically, partitions in narrow transformations can either depend on one parent (such as in the `map` operator), or a unique subset of the parent partitions that is known at design time (`coalesce`). 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, e.g., according the value of their key. In `sort`, for example, records have to be partitioned so that keys in the same range are on the same partition. Transformations with wide dependencies include `sort`, `reduceByKey`, `groupByKey`, `join`, and anything that calls the `rePartition` function.

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. They become more expensive with more data and when a greater proportion of that data has to be moved to a new partition during the shuffle. As we will discuss at length in Chapter 6, we can get a lot of performance gains out of Spark programs by doing fewer and less expensive shuffles.

The next two diagrams illustrate the difference in the dependency graph for transformations with narrow dependencies versus transformations with wide dependencies. Figure 2-2 shows 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 a transformation can still qualify as narrow if the child partitions may depend on multiple parent partitions, so long as the set of parent partitions can be determined regardless of the values of the data in the partitions.

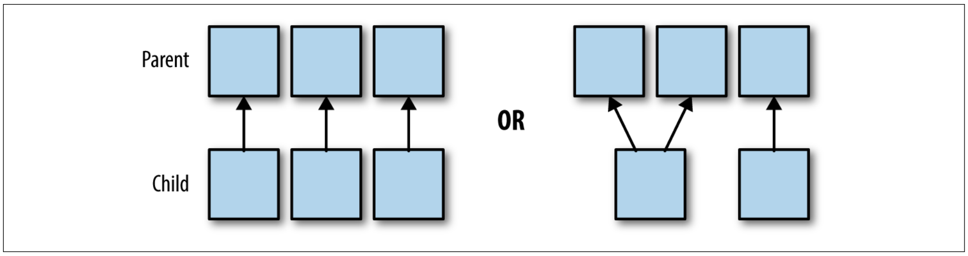


Figure 2-2. A simple diagram of dependencies between partitions for narrow transformations

Figure 2-3 shows wide dependencies between partitions. In this case the child partitions (shown at the bottom of Figure 2-3) 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 `groupByKey`, `reduceByKey`, `sort`, and `sortByKey`) follows this pattern.

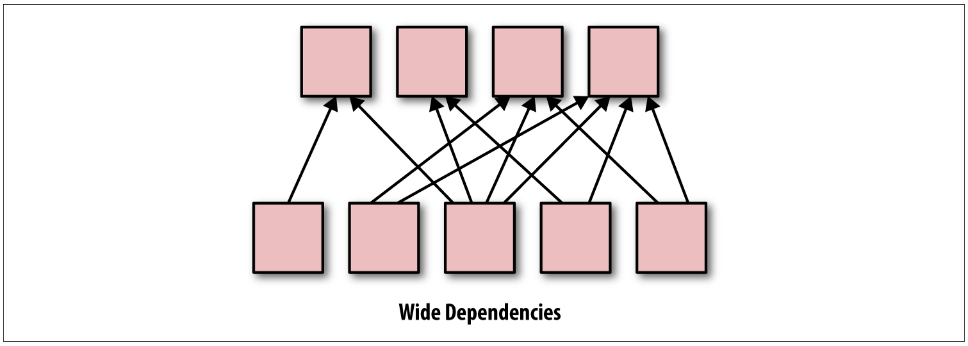


Figure 2-3. A simple diagram of dependencies between partitions for wide transformations

The join functions are a bit more complicated, since they 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 Chapter 4.

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 sends some instructions 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, in turn, 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 maximum of resources on the cluster and reserves them for the duration of the application (as long as the `SparkContext` 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 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 "Allocating Cluster Resources and Dynamic Allocation" on page 33.

The Spark Application

A Spark application corresponds to a set of Spark jobs defined by one `SparkContext` in the driver program. A Spark application begins when a `SparkContext` is started. When the `SparkContext` is started, a driver and a series of executors are started on the worker nodes of the cluster. Each executor is its own Java Virtual Machine (JVM), and an executor cannot span multiple nodes although one node may contain several executors.

The `SparkContext` determines how many resources are allotted to each executor. 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 `SparkContext` as one set of configuration parameters for running Spark jobs. These parameters are exposed in the `SparkConf` object, which is used to create a `SparkContext`. We will discuss how to use the parameters in Appendix A. Applications often, but not always, correspond to users. That is, each Spark program running on your cluster likely uses one `SparkContext`.

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

Figure 2-4 illustrates what happens when we start a `SparkContext`. First, the driver program pings the cluster manager. The cluster manager launches a number of Spark executors (JVMs shown as black boxes in the diagram) 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.

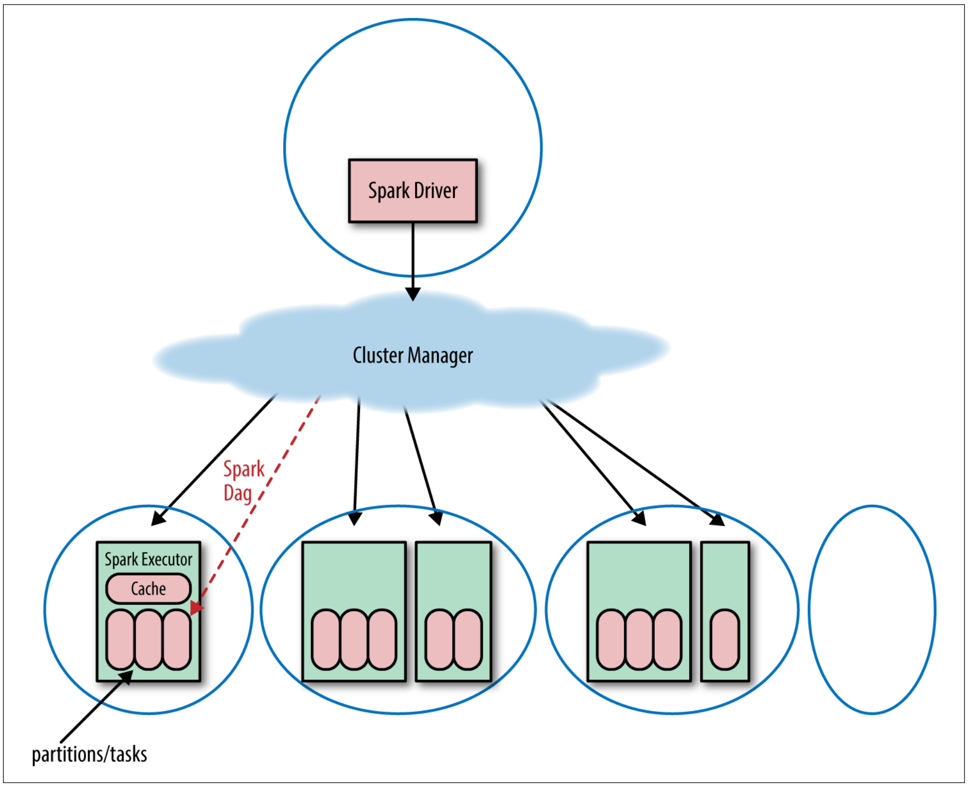


Figure 2-4. Starting a Spark application on a distributed system

Default Spark Scheduler

By default, Spark schedules jobs on 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 `SparkContext`.

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 shows a tree of the different components of a Spark application and how these correspond to the API calls. An application corresponds to starting a `SparkContext/SparkSession`. Each *application* may contain many jobs that correspond to one RDD action. Each *job* may contain several stages that correspond to each wide transformation. Each *stage* is composed of one or many tasks that 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.

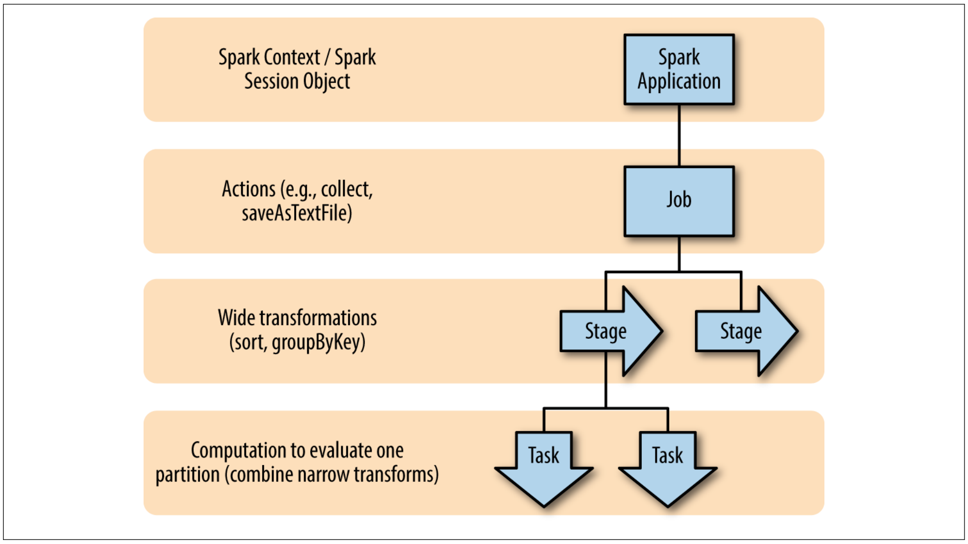

Spark Context / Spark Session Object
Spark Application
Actions (e.g., collect, saveAsTextFile)
Job
Wide transformations (sort, groupByKey)
Stage
Stage
Computation to evaluate one partition (combine narrow transforms)
Task
Task

Figure 2-5. The Spark application tree

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. 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. The `TaskScheduler` creates a graph with dependencies between partitions.[8]

Jobs

A Spark job is the highest element of Spark's execution hierarchy. Each Spark job corresponds to one action, and each action is called by the driver program of a Spark application. As we discussed in "Functions on RDDs: Transformations Versus Actions" on page 17, 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 writing to some stable storage system).

The edges of the Spark execution graph are based on dependencies between the partitions in RDD transformations (as illustrated by Figures 2-2 and 2-3). Thus, an operation that returns something other than an RDD cannot have any children. In graph theory, we would say the action forms a "leaf" in the DAG. 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. The application 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 an action is called. As mentioned previously, a job is defined by calling an action. The action may include one or several transformations, and wide transformations define the breakdown of jobs into *stages*.

Each stage corresponds to a shuffle dependency 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 whenever network communication between workers is required; for instance, in a shuffle.

These dependencies that create stage boundaries are called `ShuffleDependencies`. As we discussed in "Wide Versus Narrow Dependencies" on page 17, shuffles are caused by those wide transformations, such as `sort` or `groupByKey`, which require the data to be redistributed across the partitions. Several transformations with narrow dependencies can be grouped into one stage.

8 See *https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-TaskScheduler.html* for a more thorough description of the `TaskScheduler`.

As we saw in the word count example where we filtered stop words (Example 2-2), Spark can combine the `flatMap`, `map`, and `filter` steps into one stage since none of those transformations requires a shuffle. Thus, each executor can apply the `flatMap`, `map`, and `filter` steps consecutively in one pass of the data.

Spark keeps track of how an RDD is partitioned, so that it does not need to partition the same RDD by the same partitioner more than once. This has some interesting consequences for the DAG: the same operations on RDDs with known partitioners and RDDs without a known partitioner can result in different stage boundaries, because there is no need to shuffle an RDD with a known partition (and thus the subsequent transformations happen in the same stage). We will discuss the evaluation consequence of known partitioners in Chapter 6.

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 execute stages in parallel if they are used to compute different RDDs that 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. 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. All of the tasks in one stage 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.

Figure 2-6 shows the evaluation of a Spark job that is the result of a driver program that calls the simple Spark program shown in Example 2-3.

Example 2-3. Different types of transformations showing stage boundaries

```
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 (blue 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 rectangles), which are executed in parallel.

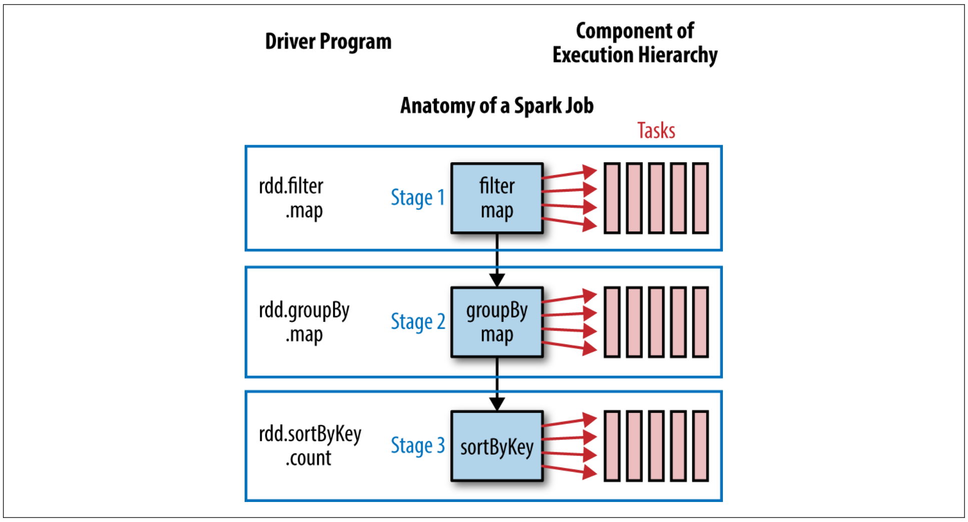


Figure 2-6. A stage diagram for the simple Spark program shown in Example 2-3

A cluster cannot necessarily run every task in parallel for each stage. Each executor has a number of cores. The number of cores per executor is configured at the application level, but likely corresponding to the physical cores on a cluster.[9] Spark can run no more tasks at once than the total number of executor cores allocated for the application. We can calculate the number of tasks from the settings from the Spark Conf as (total number of executor cores = # of cores per executor × number of executors). If there are more partitions (and thus more tasks) than the number of slots for running tasks, then the extra tasks will be allocated to the executors as the first round of tasks finish and resources are available. In most cases, all the tasks for one stage must be completed before the next stage can start. The process of distributing these tasks is done by the `TaskScheduler` and varies depending on whether the fair scheduler or FIFO scheduler is used (recall the discussion in "Default Spark Scheduler" on page 21).

9 See "Basic Spark Core Settings: How Many Resources to Allocate to the Spark Application?" on page 30 for information about configuring the number of cores and the relationship between Spark cores and the CPU on the cluster.

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 offers an innovative, efficient model of parallel computing that centers on lazily evaluated, immutable, distributed datasets, known as RDDs. Spark exposes RDDs as an interface, and RDD methods can be used without any knowledge of their implementation—but having an understanding of the details will help you write more performant code. 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.

APPENDIX A

Tuning, Debugging, and Other Things Developers Like to Pretend Don't Exist

Spark Tuning and Cluster Sizing

Recall from our discussion of Spark internals in Chapter 2 that the `SparkSession` or `SparkContext` contains the Spark configuration, which specifies how an application will be launched. Most Spark settings can only be adjusted at the application level. These configurations can have a large impact on a job's speed and chance of completing. Spark's default settings are designed to make sure that jobs can be submitted on very small clusters, and are not recommended for production.

Most often these settings will need to be changed to utilize the resources that you have available and often to allow the job to run at all. Spark provides fairly finite control of how our environment is configured, and we can often improve the performance of a job at scale by adjusting these settings. For example, in Chapter 6, we explained that out-of-memory errors on the executors was a common cause of failure for Spark jobs. While it is best to focus on the techniques presented in the preceding chapters to prevent data skew and expensive shuffles, using fewer, larger executors may also prevent failures.

Configuring a Spark job is as much an art as a science. Choosing a configuration depends on the size and setup of the data storage solution, the size of the jobs being run (how much data is processed), and the kind of jobs. For example, jobs that cache a lot of data and perform many iterative computations have different requirements than those that contain a few very large shuffles. Tuning an application also depends on the goals of your team. In some instances, if you are using shared resources, you might want to configure the job that uses the fewest resources and still succeeds.

Other times, you may want to maximize the resources available to give applications the best possible performance.

In this section, we do not aim to give a comprehensive introduction to submitting or configuring a Spark application. Instead, we want to focus on providing some context and advice about how to leverage the settings that have a significant impact on performance. In other words, we are assuming that you already have a system in which you can submit an application, but are looking for ways to adjust that system to allow your applications to run faster or run on more data.

How to Adjust Spark Settings

The `SparkContext` object (`SparkSession` in 2.0) represents your connection to the Spark application. It contains a `SparkConf` object that defines how a Spark application should be configured on your system. The `SparkConf` contains all the configurations, defaults, and environment information that govern the behavior of the Spark application. These settings are represented as key/value pairs; e.g., setting the property, `spark.executor.instances` to 5, would mean submitting a job with five executors (Spark JVMs).

You may create a `SparkConf` with the desired parameters before beginning the `SparkContext`. Some of the properties, such as the name of the application, have corresponding API calls. Otherwise, set the properties of a `SparkConf` directly with the `.set()` method, which takes as its argument arbitrary key/value pairs. To configure a Spark application differently for each submit, you may simply create an empty `SparkConf` object and supply the configurations at runtime. See the Spark documentation.[1]

The configurations for a running job can be found in the "environment" tab of the web UI.

How to Determine the Relevant Information About Your Cluster

The primary resources that the Spark application manages are CPU (number of cores) and memory. Spark requests cannot ask for more resources than are available in the environment in which they will run. Thus, it is important to understand the CPU and memory available in the environment where the job will be run. If you set up your own cluster, the answers to these questions may be obvious. But often, we are working in a system that was set up by someone else, so it is important to know how to determine the resources available to you (or what questions to ask your sys-

1 This is a good option for Spark applications designed to run in a variety of different environments, or a use case such as ours, in which we have built a web application that submits Spark jobs from within the application.

admin). The answers to these questions depend on the kind of system that you have, but generally speaking there are four primary pieces of information we have to know about our hardware:[2]

- How large can one request be? Most systems have a limit on each request, which caps the number of resources that can be made available to each executor and the driver. In *YARN cluster mode*, this is the maximum size of the YARN container. Each Spark executor and driver must "fit" within this limit. In terms of memory, the executors and driver require the amount provided, plus overhead. We cover calculating overhead in "Calculating Executor and Driver Memory Overhead" on page 31. In *YARN client mode*, the driver runs as a process on the client, so the cluster only needs to accommodate the resources required by the Spark executors and this does not apply to the driver.
- How large is each node? When determining the number of executors and the number of cores to allocate per executor, it is important to know how much memory and CPU resources there are on each node, since one executor can use resources from only one node. The memory available to each node is likely greater than or equal to one container. However, this question is still important, for determining the number of executors. For example, suppose that we have a three-node cluster with 20 GB nodes. Even if the YARN container limit is 15 GB, we can't run four executors, since the fourth executor would need to be spread across two nodes.
- How many nodes does your cluster have? How many are active? In general it is best to have at least one executor per node. Understanding how many nodes are up can also help you determine the total resources available.
- What percent of the resources are available on the system where the job will be submitted? If you are using a shared cluster environment, is the cluster busy? Will you be submitted into a queue, and how many resources does that queue have available? For a recurring job, you may have to query the YARN (or Mesos) API to determine resource burn before submitting. Often, if a Spark application requests more resources than exist in the queue into which it is submitted, but not more than exist on the whole cluster, the request will not fail, but instead will hang in a pending state. Understanding what resources are available per queue depends on what kind of scheduler your system is using. In the capacity scheduler case, each user can use a fixed percent of the resources available. In the fair scheduler case, the active applications must share resources equally. The behavior of the capacity and fair scheduler are well explained in the Spark documenta-

2 For a comprehensive answer to these questions for applications using YARN cluster mode, see this three-part post.

tion. I have also outlined how to determine that information from the YARN API in this post for details.

Basic Spark Core Settings: How Many Resources to Allocate to the Spark Application?

The `SparkSession`/`SparkContext` begins JVMs for the executors (and in YARN cluster mode, the driver). Recall that the executors run each task (in order to compute each partition) with the cores available to each executor. Furthermore, some proportion of each executor is used for computation while some is used for caching. The size of the driver, size of the executors, and the number of cores associated with each executor, are configurable from the conf and static for the duration of a Spark application. All executors are required to be the same size. Without dynamic allocation (see "Allocating Cluster Resources and Dynamic Allocation" on page 33), the number of executors is static as well. With dynamic allocation, Spark may request decommissioning of executors between stages. Although discussed in detail in Chapter 2, I think it bears repeating that the size of each executor, the driver, and the number of cores in each driver remains fixed regardless the size of your query. So, although dynamic allocation allows Spark to add an additional executor to compute a job, Spark cannot give an executor more resources when computing a particularly expensive partition, or as resources on your environment are made available.

First we will go over the meaning of each setting. Then we will weigh in on how to determine the optimal solution for parsing the resources available between the drivers and executors given the resources you have available.

Spark setting name	Meaning	Default value	Restrictions	Guidelines
spark.driver.memory	The size of the Spark driver in MB	1024 MB	In YARN cluster mode no larger than the YARN container including overhead.	A higher setting may be required if collecting large RDDs to the driver or performing many local computations.
spark.executor.memory	1024 MB	The size of the each Spark worker.	One executor + overhead cannot be larger than the limit for one request (the size of one YARN container).	Larger Spark workers may prevent out-of-memory errors, particularly if jobs require unbalanced shuffles, but may be less efficient.
spark.executor.cores	1	The number of virtual cores that will be allocated to each executor.	The number of cores available in the YARN container.	Should be around five. Scale up as resources allow.

The total memory required by all of the executors (with overhead) and the driver (with overhead) cannot be larger than the amount of memory available on the cluster. In YARN client mode, the driver does not use resources on the cluster.

Calculating Executor and Driver Memory Overhead

In YARN cluster mode and YARN client mode, both the executor memory overhead and driver memory overhead can be set manually. In both modes the executor memory overhead is set with the `spark.yarn.executor.memoryOverhead` value. In YARN cluster mode the driver memory is set with `spark.yarn.driver.memoryOverhead`, but in YARN client mode that value is called `spark.yarn.am.memoryOverhead`. In either case, the following equations govern how memory overhead is handled when these values are not set:[3]

```
memory overhead =
       Max(MEMORY_OVERHEAD_FACTOR x requested memory, MEMORY_OVERHEAD_MINIMUM).

Where MEMORY_OVERHEAD_FACTOR = 0.10 and
MEMORY_OVERHEAD_MINIMUM = 384 mb.
```

How Large to Make the Spark Driver

In general, most of the computational work of a Spark query is performed by the executors, so increasing the size of the driver rarely speeds up a computation. However, jobs may fail if they collect too much data to the driver or perform large local computations. Thus, increasing the driver memory and correspondingly the value of `spark.driver.maxResultSize` may prevent the out-of-memory errors in the driver.

> Regardless of driver memory the size of the results that can be returned to the driver are limited by the setting `spark.driver.maxResultSize`. This value bounds the total size of serialized results from all partitions being collected to the driver. The setting is used to force jobs that are likely to cause driver memory errors to fail earlier and more clearly. The default value for this setting is 1g, which is relatively small. If your job requires collecting large results and you are not competing for resources with other users, you may set the `maxResultSize` to "0", and Spark will disregard this limit.

In my experience, a good heuristic for setting the Spark driver memory is simply the lowest possible value that does not lead to memory errors in the driver, i.e., which gives the maximum possible resources to the executors.

3 See *https://www.cloudera.com/documentation/enterprise/5-6-x/topics/cdh_ig_running_spark_on_yarn.html* for a more detailed description of memory overhead and guidelines for how to configure it.

In YARN and Mesos cluster mode, the driver can be run with more cores, by setting the value of `spark.driver.cores`, and can run a multithreaded process. Otherwise, the driver requires one core.

A Few Large Executors or Many Small Executors?

We know that the total resources required by the executors and driver cannot be larger than the resources we have available, and each executor cannot request more memory or more cores than the resources allocated for one node (or container). However, this still leaves many, many options for how to allocate resources amongst the Spark workers. For example, suppose we are submitting a job to a cluster that has four 20 GB nodes with six cores each; do we create four 20 GB six-core executors, or eight 10 GB three-core executors? Good question! I spent several months trying to develop an algorithm to answer this question, and although there are some instances for which we can make an educated guess about resource allocation, finding the optimal configuration for one application on one cluster is not an exact science. Instead I hope to provide a few tips about how to recognize the consequences of either too large or too small executors in terms of either CPU or memory. Hopefully these tips will help you make an educated guess about configuring an application and help you determine how to correct a job if you see signs that it is misconfigured.

Many small executors

There are two potential downsides to using many small executors. The first has to do with the risk or running out of resources to compute a partition, as we discussed in Chapter 5. Since each partition cannot be computed on more than one executor, the size of each partition is bounded by the space they have to be computed. Thus, we risk running into memory problems, or spilling to disk if we need to shuffle, cache unbalanced data, or perform very expensive narrow transformations. If the executors have only one core, then we can run at most one task in each executor, which throws away the benefits of something like a broadcast variable, which only has to be sent to each executor (not each partition like other variables defined in the driver).

The second problem is that having too many executors may not be an efficient use of our resources. Each executor has some overhead, and there is some cost to communicating between executors even if they are on the same node. Recall from our discussion of memory overhead that the minimum overhead is just under 400 MB. Thus if we have many 1 GB executors, nearly 25 percent of the space that each executor will use on our cluster has to be used for overhead rather than computation. I think that there is a good argument to be made that if resources are available, executors should be no smaller than about four gigs, at which point overhead will be based on the memory overhead factor (a constant 10 percent of the executor).

Many large executors

Very large executors may be wasteful just because placing executors on nodes is a binning problem.[4] To use all the resources and to have the driver smaller than the size of one node, we might need to have more executors per node than one. For example, suppose that our cluster only has four very large nodes and our computation requires very little driver memory. In this case, having three very large executors and a driver that is only half the size of the executors may be wasteful, since it leaves half of the last node unused. Furthermore, very large executors may cause delays in garbage collection, since a larger heap will delay the time until a GC event is triggered and consequently GC pauses may be larger. Many cores per executor seems to lead to poor performance, due to some limitations from HDFS on handling many concurrent threads.[5] Sandy Ryza suggests that five cores per executor should be the upper limit. I have had jobs perform with a few more (6 or 7 cores), but it seems that, at the very least, assigning executors more than about seven or eight cores does not speed up performance and burns CPU resources unnecessarily. This limit on the CPU should correlate to some limitation in terms of executor memory, if you want to burn CPU and memory relatively evenly on your cluster. In other words, I have had relatively good results by determining the number of executors based on CPU resources —dividing the CPU on each node by about five—then setting memory per executor based on that number of executors.

Allocating Cluster Resources and Dynamic Allocation

Dynamic allocation is a process by which a Spark application can request and decommission executors as needed throughout the course of an application. This can lead to dramatic performance improvements especially on a busy cluster, because it allows an application to use resources as they become available and frees up those resources when jobs do not need them.

The following rules govern when Spark adds or removes executors with dynamic allocation. First, Spark requests additional executors when there are pending tasks. Second, Spark decommissions executors that have not been used to compute a job in the amount of time specified by the `spark.dynamicAllocation.executorIdleTimeout` parameter (by default, sixty seconds). With the default settings, Spark does not remove executors that contain cached data, because once an executor has been decommissioned, the cached data has to be recomputed to be used. You may change this behavior by setting `spark.dynamicAllocation.cachedExecutorIdleTimeout` to

4 For any algorithm junkies, finding the best size and number of executors is similar to the Np-Complete knapsack problem. The executors have fixed sizes and cores and have to “fit” onto the various nodes.

5 See this post by Sandy Ryza, and the data presented in this Stack Overflow post.

something other than the default: `infinity`. In this case, executors with cached data will be decommissioned if they have not been used for some amount of time.

You may configure the number of executors that Spark should start with when an application is launched with `spark.dynamicAllocation.initialExecutors`, which by default is zero. If you know that the application will be launching expensive jobs and that cluster resources are available, I would recommend increasing this. Otherwise, leaving the value at the default zero is advantageous because it means that the application can scale up resources gradually. There are also configuration values for the minimum and maximum amount of executors used during a job. I suggest setting the maximum amount to be the resources that are available to the user submitting the application on your cluster to avoid hogging the entire cluster.

Because dynamic allocation does not allow executors to change in size you still must determine the size of each executor before starting the job. My recommendation is to size the executors as you would if you were trying to use all the resources on the cluster. This will ensure that if computations are expensive and Spark requests the maximum number of executors, those resources will be well allocated. One possible exception is in the case of a very high-traffic cluster. In this case, using small executors may allow dynamic allocation to increase the number of resources used more quickly if space becomes available on the nodes in a piecemeal way.

Restrictions on dynamic allocation

Dynamic allocation can be a bit difficult to configure. In order to get dynamic allocation to work, you must:

1. Set the configuration value `spark.dynamicAllocation.enabled` to true.
2. Configure an external shuffle service on each worker. This varies based on the cluster manager, so see the Spark documentation for details.
3. Set `spark.shuffle.service.enabled` to `true`.
4. Do not provide a value for the `spark.executor.instances` parameter. Even if dynamic allocation is configured Spark will override that behavior and use the specified number of executors if this parameter is included in the conf.

In some instances, if the conf specifies using dynamic allocation but the shuffle service is misconfigured, the job will hang in a pending state, because the nodes do not have a mechanism to request executors. If you know the cluster has resources and see this behavior, make sure the shuffle service is configured on each worker and that the YARN conf contains the correct class path to the shuffle service.

Dividing the Space Within One Executor

In Figure 2-4, we suggested that executors were JVMs with some space set aside for caching and some for execution. While this is true, the division of memory usage within an executor is actually more complicated than the diagram might suggest, since the regions are not static. First, the JVM size set by the `spark.executor.memory` property does not include overhead, so Spark executors require more space on a cluster than this number would suggest. Within the executor memory, some of the space has to be reserved for Spark's internal metadata and user data structures (by default about 25%.) The remaining space on the executor, called `M` in the Spark documentation, is used for execution and storage. The execution memory is the memory required to compute a Spark transformation. The total space in the executor for both caching and execution is governed by a fixed fraction, exposed in the conf as the `spark.memory.fraction`. Out-of-memory errors during a transformation or when cached partitions are spilling to disk, is usually caused by the limitation in this combined storage and execution space. The default size of `M` is 0.6, so 60% of an executor is used for storage and execution. While it is possible to reduce the space used for internal metadata by increasing the size of `M`, doing so may be dangerous because this serves as a safeguard against out-of-memory errors caused by internal processes.

Within this `spark.memory.fraction`, which we will call `M`, some space is set aside for "storage," and the rest can be used for storage or execution. Storage in this case refers to Spark's in-memory storage of partitions, whether serialized or not. Rather than providing a fixed region for storage, Spark allows applications that do not cache anything in-memory to use the full memory fraction for execution. Within the execution space, Spark defines a region of the execution called `R` for storing cached data. The size of `R` is determined by a percentage of `M` set by the `spark.memory.storageFraction`. `R` is the space Spark will not reclaim for execution if there is cached data present. Spark enables persisting more data than fits in `R`, but allows the extra partitions to be evicted if a future task requires it.[6] In other words, to cache an RDD without allowing any partitions to be evicted from memory, all of the cached data must fit in `R`, the space determined by:

```
R = spark.executor.memory x
    spark.memory.fraction x spark.memory.storageFraction.
```

The following diagrams attempt to illustrate the relationship between `M` and `R`. Each box represents one Spark executor. The red quadrant at the bottom is `R`. The space below the overhead is `M`.

6 Prior to Spark 1.6.0, the storage and execution memory were strictly separated by the `spark.memory.storageFraction` value.

In Figure A-1, we have assumed that two different RDDs have been cached. And the partitions shown are those that are cached on this particular executor. The blue partition regardless of caching order, is the least recently used partition. Because the partitions do not take up all of the storage fraction R, a large computation (represented by the orange burst) may use the space in the storage fraction.

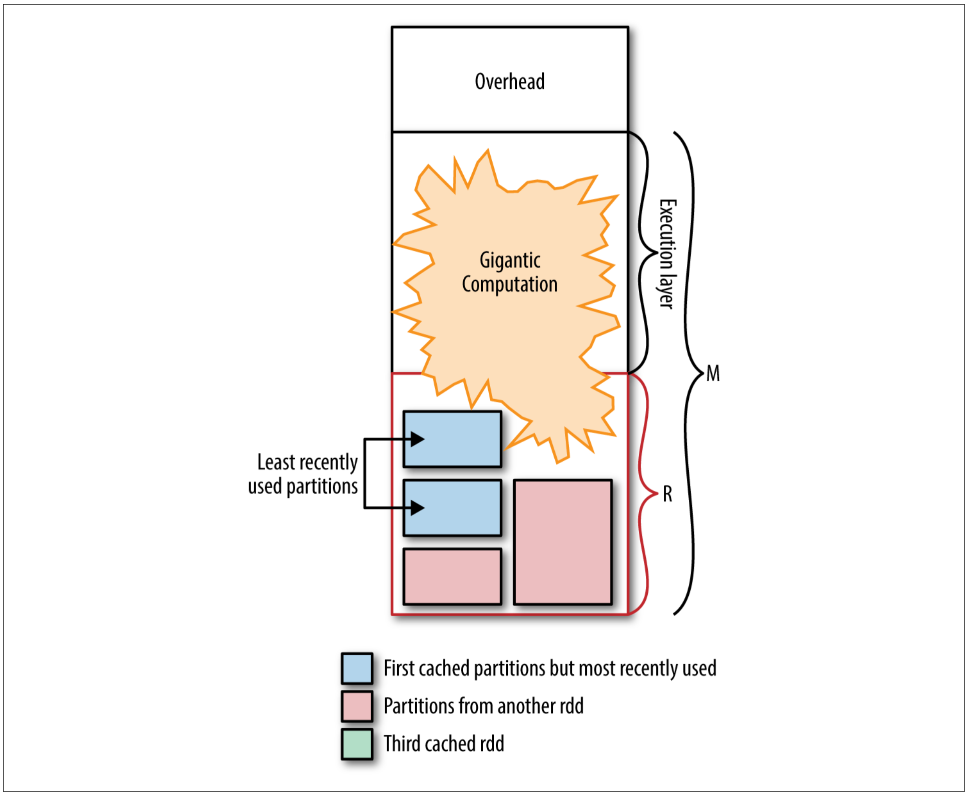

Overhead
Gigantic Computation
Execution layer
M
R
Least recently used partitions
First cached partitions but most recently used
Partitions from another rdd
Third cached rdd

Figure A-1. A single computation may use all of the space available in the memory fraction

Next, suppose that the same application included a job that cached another partition. Now all of the cached partitions take up more space than R (see that the green boxes go above the red line). This is allowed since no computation requires this space. Suppose also that the blue partition was used in this job, making the pink partition the least recently used partition. The result is shown in Figure A-2.

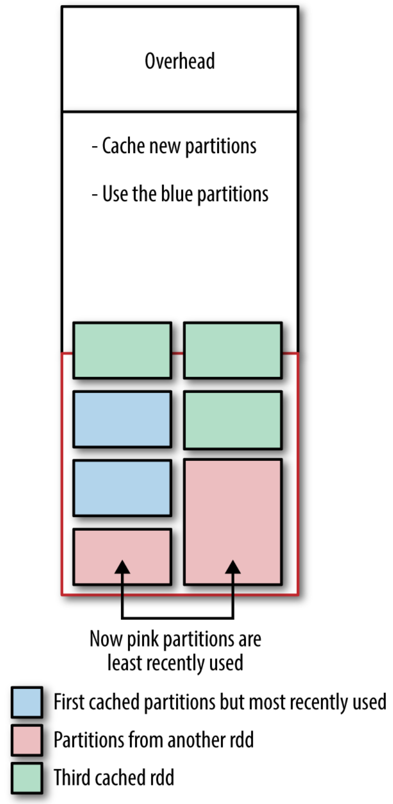

Overhead
- Cache new partitions
- Use the blue partitions
Now pink partitions are least recently used
First cached partitions but most recently used
Partitions from another rdd
Third cached rdd

Figure A-2. Cached partitions may exceed the space allocated by spark.memory.storageFraction if no computation evicts them

Now suppose that we perform a giant computation on this executor, resulting in Figure A-3. Because the cached data takes up more space than exists in the `R` region, the extra partitions will be evicted to make space for the computation. The partitions that are evicted are those that were least recently used (the pink partitions), because Spark uses Least Recently Used (LRU) caching. (See Chapter 5 for more information about LRU caching.)

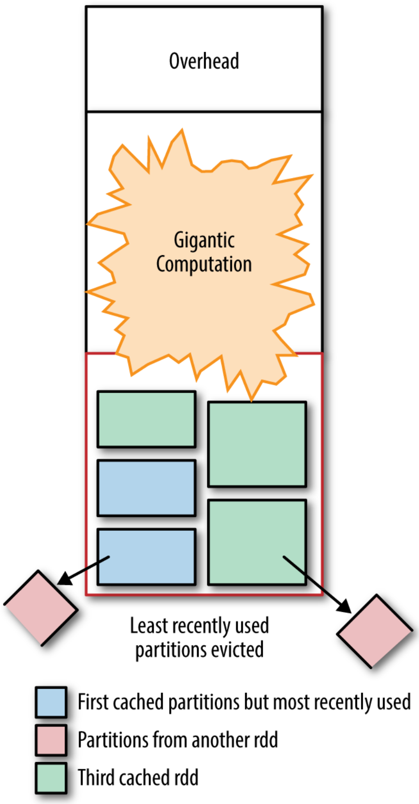
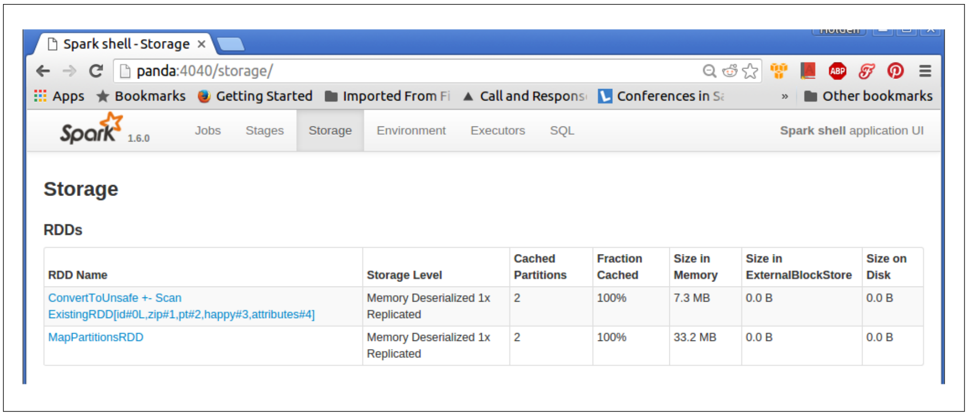

Overhead
Gigantic Computation
Least recently used partitions evicted
First cached partitions but most recently used
Partitions from another rdd
Third cached rdd

Figure A-3. A large computation may evict the cached partitions, if the storageFraction is full. The least recently used partitions are evicted first.

Adjusting the memory and storage fraction settings largely depends on the kind of computation that we want to perform. If you are not caching data, then this setting hardly matters because all of `M` will be reserved for computation anyway. However, if an application requires repeated access to an RDD and performance is improved by caching the RDD in memory, it may be useful to increase the size of the storage fraction to prevent the RDDs you needed cached from being evicted.

One way to get a feel for the size of the RDD is to write a program that caches the RDD. Then, launch the job and while it is running, look at the "Storage" page of the web UI.

In Figure A-4 we see how a cached `DataFrame` and a cached RDD appear in the web UI. The "Size In Memory" column displaces the size of the data structure in memory.



Storage

RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in ExternalBlockStore	Size on Disk
ConvertToUnsafe +- Scan ExistingRDD[id#0L,zip#1,pt#2,happy#3,attributes#4]	Memory Deserialized 1x Replicated	2	100%	7.3 MB	0.0 B	0.0 B
MapPartitionsRDD	Memory Deserialized 1x Replicated	2	100%	33.2 MB	0.0 B	0.0 B

Figure A-4. The storage tab of the web UI

Number and Size of Partitions

As we discussed in Chapter 6, Spark does not have any mechanism to set the optimal number of partitions to use. By default, when an RDD is created by reading from stable storage, the number of partitions corresponds to the splits configured in that input format (usually the parts in a MapReduce file). We can explicitly change the number of partitions using `coalesce`, `repartition`, or during wide transformations such as `reduceByKey` or `sort`. If the number of partitions is not specified for the wide transformation, Spark defaults to using the number specified by the conf value of `spark.default.parallelism`.

The default values of the `spark.default.parallelism` parameters depends on the environment that the application is running in. In YARN Cluster mode, it is the number of cores * the number of executors (in other words, the number of tasks that can be run at one time). This is the minimum value that you should use for the number of partitions, but not necessarily the optimal value.

So how many partitions should we use in a wide transformation or as the value of `spark.default.parallelism`?

Like most of my advice in this appendix, there is not a straight answer to this question. In general, increasing partitions improves performance until a certain point, when it simply creates too much overhead. At a minimum you should use as many partitions as total cores, because using fewer leaves some of the CPU resources idle. Increasing the number of partitions can also help reduce out-of-memory errors, since it means that Spark will operate on a smaller subset of the data for each executor. One strategy can be to determine the maximum size of a partition. That is, the largest a partition can be and still "fit" in the space allocated for one task and then work backward to determine the number of partitions to set (parts to divide the RDD into) so that each one is no larger than this size. Recall from Chapter 2 that each executor can run one task for each core and that one partition corresponds to one task. As we explained in "Dividing the Space Within One Executor" on page 35, the space that a Spark executor has available to compute is between the size of `M` and `M – R` depending on the amount of cached data. For example:

```
memory_for_compute (M)<
  (spark.executor.memory - over head) * spark.memory.fraction
```

And if there is cached data:

```
memory_for_compute (M - R) <
    (spark.executor.memory - overhead) x
    spark.memory.fraction x
    (1 - spark.memory.storage.fraction).
```

Assuming this space is divided equally between the tasks, then each task should not take more than:

```
memory_per_task =
    memory_for_compute / spark.executor.cores
```

Thus, we should set the number of partitions to:

```
number_of_partitions =
    size of shuffle stage / memory per task.
```

If partitions are larger than this size, then Spark may not be able to compute as many tasks concurrently, wasting CPU resources, since one task can use only one core. It may also increase the possibility of out-of-memory errors.

We can estimate the size of the stage using some information from the web UI. First, if you have cached the RDD in-memory and observed its size in the web UI, you can expect that a computation will require at least that much space, since presumably the computation requires loading the data first. We can also try to observe the shuffle stage to determine the memory cost of a shuffle. If we observe that during the shuffle stage the computation is not spilling to disk, then it is likely that each partition is fitting comfortably in-memory and we don't need to increase the number of partitions.

You can see if, and how much, tasks are spilling to disk with the web UI. To examine one stage, navigate to the "jobs" tab of the web UI while a job is running. Then click the stage that is running. There you will see the details for the stage. In the details for the stage, Spark lists a metric for the work done on each executor and for the tasks that are running. In the tasks table you will see the two last columns are for shuffle spill (memory) and shuffle (spill disk). If there are zero, then none of these tasks spilled to disk.[7]

If the job is spilling to disk it may be worth trying to estimate the size of the shuffle and trying to tune the number of partitions. Of course "the size of the shuffle" is not information we can determine concretely. Sandy Ryza suggests using the ratio between the amount of shuffle spill to memory (which appears in the UI as "Shuffle spill (memory)") and shuffle spill to disk (in the UI as "shuffle spill (disk)") and multipliying that ration by the size of the data on disk to approximate the size of the shuffle files.[8] We will elaborate on this procedure in the following section.

Shuffle spill (memory) is the amount of space that records took up in-memory *before* spilling to to disk. Shuffle spill (disk) is the space that the records took up *after* they had been spilled. Thus, the ratio between Shuffle spill (memory) and Shuffle spill (disk) is a measure of how much more space records take in-memory than on disk. I will call this the "rate of in memory expansion," which could be formalized with the following equation:

```
rate of in-memory expansion =
     Shuffle spill (memory) /Shuffle spill (disk)
```

In the web UI, we can see the total size of the shuffled write in the Initial Stages tab. This represents the size of the shuffle files written to disk.

Thus the size of those files in-memory is:

```
Size of shuffle in-memory =
     shuffle write * Shuffle spill (memory) /
     Shuffle spill (disk).
```

Again, for all the complexity of this method, it is really just a heuristic. It assumes incorrectly that each record will expand when read into memory at the same rate. There is no replacement for simply increasing the number of partitions until performance stops improving.

7 See *http://jason4zhu.blogspot.com/2015/06/roaming-through-spark-ui-and-tune-performance-upon-a-specific-use-case.html*.

8 This is Ryza's much cited blog post on Spark tuning and parallelism. It was written for an older version of Spark, so you will notice that his description of memory management differs from mine. However, most of the information, particularly the discussion of sizing partitions is still very relevant.

Serialization Options

By default Spark uses Java serialization for RDDs and Tungsten-based serialization for `DataFrames`/`Datasets`. For those who can, using `DataFrames` or `Datasets` gives you access to a much more efficient serialization layer, but if working in RDDs you can also consider using Kryo serialization.

Kryo

Like the Tungsten serializer, Kryo serialization does not natively support all of the same types that are Java serializable. The details of how Kryo works with Spark and how to extend it are already covered in *Learning Spark* (and you can find public examples as well). One might think with Tungsten, work on Spark's Kryo integration would stop, but in fact improvement to Spark's Kryo serialization are continuing. The next version of Spark is adding support for Kryo's unsafe serializer, which can be even faster than Tungsten and can be enabled by setting `spark.kryo.unsafe` to true.

Spark settings conclusion

Adjusting Spark settings can lead to huge performance improvements. However, it is time consuming and in some case provides limited gains for hours of testing. No amount of tuning would make the unbalanced shuffle presented in Chapter 6 complete on a billion rows (believe me, we tried). Because of the many variables associated with tuning an application—cluster dimensions, cluster traffic, input data size, type of computation—finding the optimal Spark configuration is difficult to do without some trial and error. However, a good understanding of what to look for in the UI, such as whether shuffles are spilling to disk or include retries, may help improve this process. Most importantly, a good knowledge of your system and the needs of your computation can help you plan the best strategy for submitting an application.

Some Additional Debugging Techniques

Debugging is an important part of the software development life cycle, and debugging with Spark has some unique considerations. The most obvious challenge of debugging Spark is that as a distributed system it can be difficult to determine which machine(s) are throwing errors, and getting access to the worker nodes may not be feasible for debugging. In addition, Spark's use of lazy evaluation can trip up developers used to more classic systems, as any logs or stack trace may at first glance point us in the wrong direction.

Debugging a system that relies on lazy evaluation requires removing the assumption that the error is necessarily directly related to the line of code that appears to trigger the error, or any of the functions it directly calls. If you are in an interactive environment, encountering an error can quickly be tracked down by adding a `take(1)` or `count` call on the parent RDDs or `DataFrames` to track down the issue more quickly. However, when debugging production code, the luxury of putting a `take(1)` or `count` may not be available, in part because it can substantially increase the computation time. Instead in those cases, it can be an important exercise to learn to tell the difference between two seemingly similar failures. Let us consider Examples A-1 and A-2 as well as the resulting stack traces in Examples A-3 and A-4, respectively.

Example A-1. Throw from inside an inner RDD

```
    val data = sc.parallelize(List(1, 2, 3))
    // Will throw an exception when forced to evaluate
    val transform1 = data.map(x => x/0)
    val transform2 = transform1.map(x => x + 1)
    transform2.collect() // Forces evaluation
```

Example A-2. Throw from inside the topmost RDD

```
    val data = sc.parallelize(List(1, 2, 3))
    val transform1 = data.map(x => x + 1)
    // Will throw an exception when forced to evaluate
    val transform2 = transform1.map(x => x/0)
    transform2.collect() // Forces evaluation
```

Example A-3. Inner failure

```
17/01/23 12:41:36 ERROR Executor: Exception in task 0.0 in stage 0.0 (TID 0)
java.lang.ArithmeticException: / by zero
    at com.highperformancespark.examples.errors.Throws$$anonfun$1
    .apply$mcII$sp(throws.scala:9)
    at com.highperformancespark.examples.errors.Throws$$anonfun$1
    .apply(throws.scala:9)
    at com.highperformancespark.examples.errors.Throws$$anonfun$1
    .apply(throws.scala:9)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$class.foreach(Iterator.scala:750)
    at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
    at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
    at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
    at scala.collection.AbstractIterator.to(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
    at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
    at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13.
    apply(RDD.scala:935)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13.
    apply(RDD.scala:935)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:99)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/01/23 12:41:36 WARN TaskSetManager: Lost task 0.0
in stage 0.0 (TID 0, localhost, executor driver):
java.lang.ArithmeticException: / by zero
    at com.highperformancespark.examples.errors.Throws$$anonfun$1
    .apply$mcII$sp(throws.scala:9)
    at com.highperformancespark.examples.errors.Throws$$anonfun$1
    .apply(throws.scala:9)
    at com.highperformancespark.examples.errors.Throws$$anonfun$1
    .apply(throws.scala:9)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$class.foreach(Iterator.scala:750)
    at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
    at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
    at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
    at scala.collection.AbstractIterator.to(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
    at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
    at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:99)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
```

```
17/01/23 12:41:36 ERROR TaskSetManager:
Task 0 in stage 0.0 failed 1 times; aborting job
org.apache.spark.SparkException: Job aborted due to stage failure:
Task 0 in stage 0.0 failed 1 times, most recent failure:
Lost task 0.0 in stage 0.0 (TID 0, localhost, executor driver):
java.lang.ArithmeticException: / by zero
    at com.highperformancespark.examples.errors.Throws$$anonfun$1.
    apply$mcII$sp(throws.scala:9)
    at com.highperformancespark.examples.errors.Throws$$anonfun$1.apply(throws.scala:9)
    at com.highperformancespark.examples.errors.Throws$$anonfun$1.apply(throws.scala:9)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$class.foreach(Iterator.scala:750)
    at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
    at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
    at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
    at scala.collection.AbstractIterator.to(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
    at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
    at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:99)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

Driver stacktrace:
  at org.apache.spark.scheduler.
  DAGScheduler.org$apache$spark$scheduler$DAGScheduler$$
  failJobAndIndependentStages(DAGScheduler.scala:1435)
  at org.apache.spark.scheduler.DAGScheduler$$anonfun$abortStage$1
  .apply(DAGScheduler.scala:1423)
  at org.apache.spark.scheduler.DAGScheduler$$anonfun$abortStage$1
  .apply(DAGScheduler.scala:1422)
  at scala.collection.mutable.ResizableArray$class.foreach(ResizableArray.scala:59)
  at scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:48)
  at org.apache.spark.scheduler.DAGScheduler.abortStage(DAGScheduler.scala:1422)
  at org.apache.spark.scheduler
  .DAGScheduler$$anonfun$handleTaskSetFailed$1.apply(DAGScheduler.scala:802)
  at org.apache.spark.scheduler
  .DAGScheduler$$anonfun$handleTaskSetFailed$1.apply(DAGScheduler.scala:802)
  at scala.Option.foreach(Option.scala:257)
  at org.apache.spark.scheduler.DAGScheduler.
  handleTaskSetFailed(DAGScheduler.scala:802)
  at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.
  doOnReceive(DAGScheduler.scala:1650)
  at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop
  .onReceive(DAGScheduler.scala:1605)
  at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop
  .onReceive(DAGScheduler.scala:1594)
  at org.apache.spark.util.EventLoop$$anon$1.run(EventLoop.scala:48)
  at org.apache.spark.scheduler.DAGScheduler.runJob(DAGScheduler.scala:628)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1918)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1931)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1944)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1958)
  at org.apache.spark.rdd.RDD$$anonfun$collect$1.apply(RDD.scala:935)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
  at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
  at org.apache.spark.rdd.RDD.collect(RDD.scala:934)
  at com.highperformancespark.examples.errors.Throws$.throwInner(throws.scala:11)
  ... 43 elided
Caused by: java.lang.ArithmeticException: / by zero
  at com.highperformancespark.examples.errors.Throws$$anonfun$1
  .apply$mcII$sp(throws.scala:9)
  at com.highperformancespark.examples.errors.Throws$$anonfun$1.apply(throws.scala:9)
  at com.highperformancespark.examples.errors.Throws$$anonfun$1.apply(throws.scala:9)
  at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
  at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
  at scala.collection.Iterator$class.foreach(Iterator.scala:750)
  at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
  at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
  at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
  at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
  at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
  at scala.collection.AbstractIterator.to(Iterator.scala:1202)
  at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
  at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
  at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
  at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
  at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13.apply(RDD.scala:935)
  at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13.apply(RDD.scala:935)
  at org.apache.spark.SparkContext$$anonfun$runJob$5.apply(SparkContext.scala:1944)
  at org.apache.spark.SparkContext$$anonfun$runJob$5.apply(SparkContext.scala:1944)
  at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
  at org.apache.spark.scheduler.Task.run(Task.scala:99)
  at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
  ... 1 more
```

Example A-4. Throw outer exception

```
17/01/23 12:45:27 ERROR Executor: Exception in task 0.0 in stage 1.0 (TID 1)
java.lang.ArithmeticException: / by zero
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply$mcII$sp(throws.scala:17)
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply(throws.scala:17)
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply(throws.scala:17)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$class.foreach(Iterator.scala:750)
    at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
    at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
    at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
    at scala.collection.AbstractIterator.to(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
    at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
    at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:99)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/01/23 12:45:27 WARN TaskSetManager: Lost task 0.0 in stage 1.0
(TID 1, localhost, executor driver):
java.lang.ArithmeticException: /
by zero
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply$mcII$sp(throws.scala:17)
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply(throws.scala:17)
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply(throws.scala:17)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$class.foreach(Iterator.scala:750)
    at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
    at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
    at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
    at scala.collection.AbstractIterator.to(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
    at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
    at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:99)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

17/01/23 12:45:27 ERROR TaskSetManager: Task 0 in stage 1.0 failed 1 times;
aborting job
org.apache.spark.SparkException: Job aborted due to stage failure:
Task 0 in stage 1.0 failed 1 times, most recent failure:
Lost task 0.0 in stage 1.0 (TID 1, localhost, executor driver):
java.lang.ArithmeticException: / by zero
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply$mcII$sp(throws.scala:17)
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply(throws.scala:17)
    at com.highperformancespark.examples.errors.Throws$$anonfun$4
    .apply(throws.scala:17)
    at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
    at scala.collection.Iterator$class.foreach(Iterator.scala:750)
    at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
    at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
    at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
    at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
    at scala.collection.AbstractIterator.to(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
    at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
    at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
    at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13
    .apply(RDD.scala:935)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.SparkContext$$anonfun$runJob$5
    .apply(SparkContext.scala:1944)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:99)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

Driver stacktrace:
  at org.apache.spark.scheduler
  .DAGScheduler.org$apache$spark$scheduler$DAGScheduler$$failJobAndIndependentStages
  (DAGScheduler.scala:1435)
  at org.apache.spark.scheduler.DAGScheduler$$anonfun$abortStage$1
  .apply(DAGScheduler.scala:1423)
  at org.apache.spark.scheduler.DAGScheduler$$anonfun$abortStage$1
  .apply(DAGScheduler.scala:1422)
  at scala.collection.mutable.ResizableArray$class.foreach(ResizableArray.scala:59)
  at scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:48)
  at org.apache.spark.scheduler.DAGScheduler.abortStage(DAGScheduler.scala:1422)
  at org.apache.spark.scheduler.DAGScheduler$$anonfun$handleTaskSetFailed$1
  .apply(DAGScheduler.scala:802)
  at org.apache.spark.scheduler.DAGScheduler$$anonfun$handleTaskSetFailed$1
  .apply(DAGScheduler.scala:802)
  at scala.Option.foreach(Option.scala:257)
  at org.apache.spark.scheduler.DAGScheduler
  .handleTaskSetFailed(DAGScheduler.scala:802)
  at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop
  .doOnReceive(DAGScheduler.scala:1650)
  at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop
  .onReceive(DAGScheduler.scala:1605)
  at org.apache.spark.scheduler.DAGSchedulerEventProcessLoop
  .onReceive(DAGScheduler.scala:1594)
  at org.apache.spark.util.EventLoop$$anon$1.run(EventLoop.scala:48)
  at org.apache.spark.scheduler.DAGScheduler.runJob(DAGScheduler.scala:628)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1918)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1931)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1944)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1958)
  at org.apache.spark.rdd.RDD$$anonfun$collect$1.apply(RDD.scala:935)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
  at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
  at org.apache.spark.rdd.RDD.collect(RDD.scala:934)
  at com.highperformancespark.examples.errors.Throws$.throwOuter(throws.scala:18)
  ... 43 elided
Caused by: java.lang.ArithmeticException: / by zero
  at com.highperformancespark.examples.errors
  .Throws$$anonfun$4.apply$mcII$sp(throws.scala:17)
  at com.highperformancespark.examples.errors
  .Throws$$anonfun$4.apply(throws.scala:17)
  at com.highperformancespark.examples.errors
  .Throws$$anonfun$4.apply(throws.scala:17)
  at scala.collection.Iterator$$anon$11.next(Iterator.scala:370)
  at scala.collection.Iterator$class.foreach(Iterator.scala:750)
  at scala.collection.AbstractIterator.foreach(Iterator.scala:1202)
  at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
  at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
  at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
  at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:295)
  at scala.collection.AbstractIterator.to(Iterator.scala:1202)
  at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:287)
  at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1202)
  at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:274)
  at scala.collection.AbstractIterator.toArray(Iterator.scala:1202)
  at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13.apply(RDD.scala:935)
  at org.apache.spark.rdd.RDD$$anonfun$collect$1$$anonfun$13.apply(RDD.scala:935)
  at org.apache.spark.SparkContext$$anonfun$runJob$5.apply(SparkContext.scala:1944)
  at org.apache.spark.SparkContext$$anonfun$runJob$5.apply(SparkContext.scala:1944)
  at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
  at org.apache.spark.scheduler.Task.run(Task.scala:99)
  at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
  ... 1 more
```

These two stack traces contain a lot of information, but most of it isn't useful to debugging our error. Since the error is coming from inside of our worker we can mostly ignore the information about the driver stack trace and instead look at the exceptions reported under `17/01/23 12:41:36 ERROR Executor: Exception in task 0.0 in stage 0.0 (TID 0)`.

A common mistake is looking at the driver stack trace when the error is reported from the executor. In that case it can seem like the error is on the line of your action, whereas the root cause lies elsewhere (as in both of the preceding examples).

By looking at the error reported from the executor you can see the line number associated with the function that failed. The rest of the exception isn't normally required unless you happen to be working with `mapPartitions` and returning custom iterators, as the iterator chaining is taken care inside of Spark.

The exception is reported twice (once as a warning and then as an error) since Spark retries a partition on failures. If you had many partitions with errors this could be reported far more than twice.

You can also see these exceptions in the Spark web UI logs if your application hasn't exited.

Determining what statement threw the error can be tricky, especially with anonymous inner functions. If you instead updated Examples A-1 and A-2 to use explicit function names (e.g., Examples A-5 and A-6) you can more easily find out what's going on. The resulting stack trace now contains the function name in question (e.g., `at com.highperformancespark.examples.errors.Throws$.divZero(throws.scala:26)`).

Example A-5. Refactored helper functions

```
def add1(x: Int): Int = {
  x + 1
}

def divZero(x: Int): Int = {
  x / 0
}
```

Example A-6. Refactored throw examples to use helper functions

```
def add1(x: Int): Int = {
  x + 1
}

def divZero(x: Int): Int = {
  x / 0
}
```

You may notice that even though the underlying "cause" of the error is a division by zero (or `java.lang.ArithmeticException`), the top-level exception is wrapped by `org.apache.spark.SparkException`. To access the underlying exception you can use `getCause`.

Not all exceptions will be wrapped in `org.apache.spark.SparkException`. When you attempt to compute an RDD backed by a nonexistent Hadoop input, you get a much simpler stack trace (as in Example A-7) directly returning the underlying exception.

Example A-7. Exception when trying to load nonexistent input

```
org.apache.hadoop.mapred.InvalidInputException:
Input path does not exist: file:/doesnotexist.txt
  at org.apache.hadoop.mapred.FileInputFormat.
  singleThreadedListStatus(FileInputFormat.java:285)
  at org.apache.hadoop.mapred.FileInputFormat.listStatus(FileInputFormat.java:228)
  at org.apache.hadoop.mapred.FileInputFormat.getSplits(FileInputFormat.java:313)
  at org.apache.spark.rdd.HadoopRDD.getPartitions(HadoopRDD.scala:202)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:252)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:250)
  at scala.Option.getOrElse(Option.scala:121)
  at org.apache.spark.rdd.RDD.partitions(RDD.scala:250)
  at org.apache.spark.rdd.MapPartitionsRDD.getPartitions(MapPartitionsRDD.scala:35)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:252)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:250)
  at scala.Option.getOrElse(Option.scala:121)
  at org.apache.spark.rdd.RDD.partitions(RDD.scala:250)
  at org.apache.spark.rdd.MapPartitionsRDD.getPartitions(MapPartitionsRDD.scala:35)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:252)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:250)
  at scala.Option.getOrElse(Option.scala:121)
  at org.apache.spark.rdd.RDD.partitions(RDD.scala:250)
  at org.apache.spark.rdd.MapPartitionsRDD.getPartitions(MapPartitionsRDD.scala:35)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:252)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:250)
  at scala.Option.getOrElse(Option.scala:121)
  at org.apache.spark.rdd.RDD.partitions(RDD.scala:250)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1958)
  at org.apache.spark.rdd.RDD$$anonfun$collect$1.apply(RDD.scala:935)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
  at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
  at org.apache.spark.rdd.RDD.collect(RDD.scala:934)
  at com.highperformancespark.examples.errors.Throws$
  .nonExistentInput(throws.scala:47)
  ... 43 elided
```

Debugging driver out-of-memory exceptions can be challenging, and it can be difficult to know exactly what operation caused the failure. While you should already be weary of `collect` statements, it is important to remember other operations like `countByKey` can potentially return unbounded results to the driver program. In Spark ML and MLlib, some of the models require bringing back a large amount of data to the driver program—in that event, the easiest solution may be trying a different model.

It can be difficult to predict how much memory a driver program will use, or track down the source of high memory usage. Furthermore, the process for assigning memory to the driver program depends on how you are submitting the Spark application. When launching an application with `spark-submit` in “client mode” (or scripts based on it like `spark-shell` or `pyspark`), the driver JVM is started *before* the `spark-defaults.conf` can be parsed and before the `SparkContext` is created. In this case, you must set the driver memory in *spark-env.sh* or with the `--driver-memory` to `spark-submit`.

When launching Spark from Python without `spark-submit` or any of the helpers specifying the JVM driver size or any driver-side JVM configurations on the conf object will not be respected even though the JVM has technically not started, the JVM driver is started using the traditional `spark-submit` under the hood. Instead you can configure the shell environment variable `PYSPARK_SUBMIT_ARGS` (which defaults to `pyspark-shell`) as needed.

While not considered classic debugging techniques, resolving stragglers or otherwise imbalanced partitioning is very much related and discussed in Chapter 5.

Out of Disk Space Errors

Out of disk space errors can be surprising, but are common in clusters with small amounts of disk space. Sometimes disk space errors are caused by long-running shell environments where RDDs are created at the top scope and are never garbage collected. Spark writes the output of its shuffle operations to files on the disk of the workers in the Spark local dir. These files are only cleaned up when an RDD is garbage collected, which if the amount of memory assigned to the driver program is large can take quite some time. One solution is to explicitly trigger garbage collection (assuming the RDDs have gone out of scope); if the DAG is getting too long, checkpointing can help make the RDDs available for garbage collection.

Logging

Logging is an important part of debugging your application, and in a distributed system depending on `println` is probably not going to cut it. Spark uses the `log4j` through `sl4j` logging mechanism internally, so `log4j` can be a good mechanism for our application to log with as well since we can be relatively assured it is already set up.

Early (pre-2.0) versions of Spark exposed their logging API, which is built on top of `log4j`. This is becoming private in 2.0 and beyond, and doesn't offer much functionality that isn't available directly in `log4j`, so you should instead directly use the `log4j` logger. The author herself is guilty of having accessed the internal logging APIs.[9]

You can get access to similar functionality as the internal logging used inside of Spark through typesafe's `scalalogging` package. This package offers a trait called `LazyLogging`, which uses macros to rewrite `logger.debug(xyz)` to the equivalent, but behind a guard checking the log level. A simple example of debugging logging is wanting to be able to see what elements are being filtered out. You can add logging to the previous Example 5-16 example, resulting in Example A-9. You must also include a logging library in your build (we used Example A-8).

Example A-8. Add scalalogging to build

```
"com.typesafe.scala-logging" %% "scala-logging" % "3.5.0",
```

Example A-9. Logged broadcast of a hashset of invalid panda locations to filter out

```
val invalid = HashSet() ++ invalidPandas
val invalidBroadcast = sc.broadcast(invalid)
def keepPanda(pandaId: Long) = {
  if (invalidBroadcast.value.contains(pandaId)) {
    logger.debug(s"Invalid panda ${pandaId} discovered")
    false
  } else {
    true
  }
}
input.filter{panda => keepPanda(panda.id)}
```

Sometimes it's handy to have the logs be clearer about what RDD/partition ID/attempt number is being processed. In that case you can look to the `TaskContext` (in both JVM and Python lands) to get this information.

Configuring logging

Spark uses `log4j` for its JVM logging, and even inside of Python or R, much of the logging information is generated from inside the JVM. When running inside of the

9 And she has repented for her sins in writing this warning.

interactive shells, one of the first messages contains instructions for configuring the log level in an interactive mode:

```
To adjust logging level use sc.setLogLevel(newLevel).
For SparkR, use setLogLevel(newLevel).
```

In addition to setting the log level using the SparkContext, Spark has a *conf/log4j.properties.template* file that can be adjusted to change log level and log outputs. Simple copy *conf/log4j.properties.template* to *conf/log4j.properties* and update any required properties, e.g. if you find Spark too verbose you may wish to set the main logging to `ERROR` as done in Example A-10.

Example A-10. log4j.properties

```
log4j.logger.org.apache.spark.repl.Main=ERROR
```

You can configure log levels for different loggers to different levels (for example, the default Spark *log4j.properties.template* configures Spark logging to `WARN` and Parquet to `ERROR`. This is especially useful when your own application also uses `log4j` logging.

If *log4j.properties* doesn't suit your needs you can ship a custom *log4j.xml* file and provide it to the executors (either by including it in your JAR or with `--files`) and then add `-Dlog4j.configuration=log4j.xm` to `spark.executor.extraJavaOptions` so the executors pick it up.

Accessing logs

If your application is actively running, the Spark web UI provides an easy way to get access to the logs of the different workers. Once your application has finished, getting the logs depends more on the deployment mechanism used.

For YARN deployments with log aggregation, the `yarn logs` command can be used to fetch the logs. If you don't have log aggregation enabled, you can still access the logs by keeping them on the worker nodes for a fixed period of time with the `yarn.nodemanager.delete.debug-delay-sec` configuration property.

In addition to copying the logs back by hand, Spark has an optional "Spark History Server," which can provide a Spark UI for finished jobs.

Attaching debuggers

While log files, accumulators, and metrics can all help debug your application, sometimes what you really want is a nice IDE interface to debug with in Spark. The simplest way to do this can be running Spark in local mode and simply attaching a debugger to your local JVM. Unfortunately not all problems on real clusters can be reproduced in local mode—in that case you can set up debugging using JDWP (Java Debug Wire Protocol) to debug against a remote cluster.

Using `spark.executor.extraJavaOptions` (for the executor) or `--driver-java-options` for the driver, add the JVM parameters `-agentlib: jdwp= transport=dt_socket,server=y,address=[debugport]` to launch.

Once you have JDWP set up on the worker or driver, the rest depends on the specific IDE you are using. IBM has a guide on how to do remote debugging with eclipse and IntelliJ is covered in this Stack Overflow answer.

Remote debugging in Python requires modifying your code to start the debugging library. Multiple options exists for this, including integrated IDE debugging with Jetbrains and Eclipse, to more basic remote tracing with rpdb and more listed on the Python wiki. Regardless of which specific option you use, you can use a broadcast variable to ensure that all of the worker nodes receive the code to start up the remote Python debugging interface.

Adding `-mtrace` to your driver/worker Python path will not work due to hardcoded assumptions in PySpark about the arguments.

Debugging in notebooks

Depending on your notebook, logging information can be more difficult to keep track of. When working in Jupyter with an IPython kernel, the error messages reported only include the Java stack trace and often miss the important information from the Python interpreter itself. In this case, you will need to look at the console from which you launched Jupyter.

Launching Spark from within a notebook can have an unexpected impact on how the configuration is handled. The biggest difference can come from handling options for the driver configurations. When working in a hosted environment, such as Databricks cloud, IBM Data Science Experience, or Microsoft's Azure hosted Spark, it's important to do the configuration through the provided mechanism.

Python debugging

PySpark has some additional considerations; the architecture introduced in Figure 7-1 means an extra level of complexity is present. The most obvious manifestation of this comes from looking at the difference between the error messages in PySpark and error messages for the same in Scala (e.g., Examples A-3 and A-13). In addition to the extra level of indirection introduced, resource contention between the different programs involved can be a source of errors as well.

Lazy evaluation in PySpark becomes even a little more complicated than in Scala. To further confuse the source of the error, evaluation of PySpark RDDs are chained together to reduce the number of round trips of the data between Python and the JVM. Let's update Examples A-1 and A-2 to Python (giving us Examples A-11 and A-12) and examine the resulting error messages in Examples A-13 and A-14.

Example A-11. Throw from inside an inner RDD (Python)

```
    data = sc.parallelize(range(10))
    transform1 = data.map(lambda x: x / 0)
    transform2 = transform1.map(lambda x: x + 1)
    transform2.count()
```

Example A-12. Throw from inside the topmost RDD (Python)

```
    data = sc.parallelize(range(10))
    transform1 = data.map(lambda x: x + 1)
    transform2 = transform1.map(lambda x: x / 0)
    transform2.count()
```

Example A-13. Inner failure error message (Python)

```
[Stage 0:>                                                          (0 + 0) /
4]17/02/28 22:28:58 ERROR Executor:
Exception in task 3.0 in stage 0.0 (TID 3)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.
    read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1.
    <init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:28:58 ERROR Executor: Exception in task 2.0 in stage 0.0 (TID 2)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:28:58 ERROR Executor: Exception in task 1.0 in stage 0.0 (TID 1)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:28:58 ERROR Executor: Exception in task 0.0 in stage 0.0 (TID 0)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1
    .read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:28:58 WARN TaskSetManager:
Lost task 0.0 in stage 0.0 (TID 0, localhost, executor driver):
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
```

```
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

17/02/28 22:28:58 ERROR Executor: Exception in task 0.1 in stage 0.0 (TID 7)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:28:58 ERROR TaskSetManager: Task 0 in stage 0.0 failed 2 times;
aborting job
17/02/28 22:28:58 ERROR Executor: Exception in task 2.1 in stage 0.0 (TID 5)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner
    .run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:28:58 ERROR Executor: Exception in task 3.1 in stage 0.0 (TID 6)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:28:58 ERROR Executor: Exception in task 1.1 in stage 0.0 (TID 4)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "high_performance_pyspark/bad_pyspark.py", line 48, in throwInner
    transform2.count()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in count
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1031, in sum
    return self.mapPartitions(lambda x: [sum(x)]).fold(0, operator.add)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 905, in fold
    vals = self.mapPartitions(func).collect()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 808, in collect
    port = self.ctx._jvm.PythonRDD.collectAndServe(self._jrdd.rdd())
  File "/home/holden/repos/spark/python/lib/py4j-0.10.4-src.zip/py4j/java_gateway.py",
  line 1133, in __call__
  File "/home/holden/repos/spark/python/pyspark/sql/utils.py", line 63, in deco
    return f(*a, **kw)
  File "/home/holden/repos/spark/python/lib/py4j-0.10.4-src.zip/py4j/protocol.py",
  line 319, in get_return_value
py4j.protocol.Py4JJavaError:
An error occurred while calling z:org.apache.spark.api.python.PythonRDD
.collectAndServe.
: org.apache.spark.SparkException: Job aborted due to stage failure:
Task 0 in stage 0.0 failed 2 times, most recent failure:
```

```
Lost task 0.1 in stage 0.0 (TID 7, localhost, executor driver):
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

Driver stacktrace:
    at org.apache.spark.scheduler
    .DAGScheduler.org$apache$spark$scheduler$DAGScheduler$$
    failJobAndIndependentStages(DAGScheduler.scala:1487)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$abortStage$1.apply(DAGScheduler.scala:1475)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$abortStage$1.apply(DAGScheduler.scala:1474)
    at scala.collection.mutable.
    ResizableArray$class.foreach(ResizableArray.scala:59)
    at scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:48)
    at org.apache.spark.scheduler.DAGScheduler
    .abortStage(DAGScheduler.scala:1474)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$handleTaskSetFailed$1.apply(DAGScheduler.scala:803)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$handleTaskSetFailed$1.apply(DAGScheduler.scala:803)
    at scala.Option.foreach(Option.scala:257)
    at org.apache.spark.scheduler.DAGScheduler
    .handleTaskSetFailed(DAGScheduler.scala:803)
    at org.apache.spark.scheduler
    .DAGSchedulerEventProcessLoop.doOnReceive(DAGScheduler.scala:1702)
    at org.apache.spark.scheduler
    .DAGSchedulerEventProcessLoop.onReceive(DAGScheduler.scala:1657)
    at org.apache.spark.scheduler
    .DAGSchedulerEventProcessLoop.onReceive(DAGScheduler.scala:1646)
    at org.apache.spark.util.EventLoop$$anon$1.run(EventLoop.scala:48)
    at org.apache.spark.scheduler.DAGScheduler.runJob(DAGScheduler.scala:628)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2011)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2032)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2051)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2076)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1.apply(RDD.scala:936)
    at org.apache.spark.rdd.RDDOperationScope$
    .withScope(RDDOperationScope.scala:151)
    at org.apache.spark.rdd.RDDOperationScope$
    .withScope(RDDOperationScope.scala:112)
    at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
    at org.apache.spark.rdd.RDD.collect(RDD.scala:935)
    at org.apache.spark.api.python.PythonRDD$
    .collectAndServe(PythonRDD.scala:458)
    at org.apache.spark.api.python.PythonRDD
    .collectAndServe(PythonRDD.scala)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl
    .invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl
    .invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at py4j.reflection.MethodInvoker.invoke(MethodInvoker.java:244)
    at py4j.reflection.ReflectionEngine.invoke(ReflectionEngine.java:357)
    at py4j.Gateway.invoke(Gateway.java:280)
    at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.java:132)
    at py4j.commands.CallCommand.execute(CallCommand.java:79)
    at py4j.GatewayConnection.run(GatewayConnection.java:214)
    at java.lang.Thread.run(Thread.java:745)
Caused by: org.apache.spark.api.python.PythonException:
Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 46, in <lambda>
    transform1 = data.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1
    .read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    ... 1 more
```

Example A-14. Outer failure error message (Python)

```
17/02/28 22:29:21 ERROR Executor: Exception in task 1.0 in stage 1.0 (TID 9)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
```

```
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1
    .read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:29:21 WARN TaskSetManager:
Lost task 1.0 in stage 1.0 (TID 9, localhost, executor driver):
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

17/02/28 22:29:21 ERROR Executor: Exception in task 0.0 in stage 1.0 (TID 8)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:29:21 ERROR Executor: Exception in task 3.0 in stage 1.0 (TID 11)
```

```
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:29:21 ERROR Executor: Exception in task 1.1 in stage 1.0 (TID 12)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:29:21 ERROR TaskSetManager:
Task 1 in stage 1.0 failed 2 times; aborting job
17/02/28 22:29:21 ERROR Executor: Exception in task 2.0 in stage 1.0 (TID 10)
org.apache.spark.api.python.PythonException: Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1
    .read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
17/02/28 22:29:21 WARN TaskSetManager:
Lost task 0.1 in stage 1.0 (TID 13, localhost, executor driver):
TaskKilled (killed intentionally)
17/02/28 22:29:21 WARN TaskSetManager:
Lost task 3.1 in stage 1.0 (TID 14, localhost, executor driver):
TaskKilled (killed intentionally)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "high_performance_pyspark/bad_pyspark.py", line 33, in throwOuter
    transform2.count()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in count
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1031, in sum
    return self.mapPartitions(lambda x: [sum(x)]).fold(0, operator.add)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 905, in fold
    vals = self.mapPartitions(func).collect()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 808, in collect
    port = self.ctx._jvm.PythonRDD.collectAndServe(self._jrdd.rdd())
  File "/home/holden/repos/spark/python/lib/py4j-0.10.4-src.zip/py4j/java_gateway.py",
  line 1133, in __call__
  File "/home/holden/repos/spark/python/pyspark/sql/utils.py", line 63, in deco
    return f(*a, **kw)
  File "/home/holden/repos/spark/python/lib/py4j-0.10.4-src.zip/py4j/protocol.py",
  line 319, in get_return_value
py4j.protocol.Py4JJavaError:
An error occurred while calling z:org.apache.spark.api.python.PythonRDD
.collectAndServe.
: org.apache.spark.SparkException:
Job aborted due to stage failure: Task 1 in stage 1.0 failed 2 times,
most recent failure: Lost task 1.1 in stage 1.0 (TID 12, localhost,
executor driver): org.apache.spark.api.python.PythonException:
Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main
    process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
```

```
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1
    .read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)

Driver stacktrace:
    at org.apache.spark.scheduler
    .DAGScheduler.org$apache$spark$scheduler$DAGScheduler$$
    failJobAndIndependentStages(DAGScheduler.scala:1487)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$abortStage$1.apply(DAGScheduler.scala:1475)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$abortStage$1.apply(DAGScheduler.scala:1474)
    at scala.collection.mutable.ResizableArray$class
    .foreach(ResizableArray.scala:59)
    at scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:48)
    at org.apache.spark.scheduler
    .DAGScheduler.abortStage(DAGScheduler.scala:1474)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$handleTaskSetFailed$1.apply(DAGScheduler.scala:803)
    at org.apache.spark.scheduler
    .DAGScheduler$$anonfun$handleTaskSetFailed$1.apply(DAGScheduler.scala:803)
    at scala.Option.foreach(Option.scala:257)
    at org.apache.spark.scheduler
    .DAGScheduler.handleTaskSetFailed(DAGScheduler.scala:803)
    at org.apache.spark.scheduler
    .DAGSchedulerEventProcessLoop.doOnReceive(DAGScheduler.scala:1702)
    at org.apache.spark.scheduler
    .DAGSchedulerEventProcessLoop.onReceive(DAGScheduler.scala:1657)
    at org.apache.spark.scheduler
    .DAGSchedulerEventProcessLoop.onReceive(DAGScheduler.scala:1646)
    at org.apache.spark.util.EventLoop$$anon$1.run(EventLoop.scala:48)
    at org.apache.spark.scheduler.DAGScheduler.runJob(DAGScheduler.scala:628)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2011)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2032)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2051)
    at org.apache.spark.SparkContext.runJob(SparkContext.scala:2076)
    at org.apache.spark.rdd.RDD$$anonfun$collect$1
    .apply(RDD.scala:936)
    at org.apache.spark.rdd.RDDOperationScope$
    .withScope(RDDOperationScope.scala:151)
    at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
    at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
    at org.apache.spark.rdd.RDD.collect(RDD.scala:935)
    at org.apache.spark.api.python.PythonRDD$
    .collectAndServe(PythonRDD.scala:458)
    at org.apache.spark.api.python.PythonRDD.collectAndServe(PythonRDD.scala)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl
    .invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl
    .invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at py4j.reflection.MethodInvoker.invoke(MethodInvoker.java:244)
    at py4j.reflection.ReflectionEngine.invoke(ReflectionEngine.java:357)
    at py4j.Gateway.invoke(Gateway.java:280)
    at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.java:132)
    at py4j.commands.CallCommand.execute(CallCommand.java:79)
    at py4j.GatewayConnection.run(GatewayConnection.java:214)
    at java.lang.Thread.run(Thread.java:745)
Caused by: org.apache.spark.api.python.PythonException:
Traceback (most recent call last):
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 180, in main process()
  File "/home/holden/repos/spark/python/lib/pyspark.zip/pyspark/worker.py",
  line 175, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 2406, in pipeline_func
    return func(split, prev_func(split, iterator))
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 345, in func
    return f(iterator)
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <lambda>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "/home/holden/repos/spark/python/pyspark/rdd.py", line 1040, in <genexpr>
    return self.mapPartitions(lambda i: [sum(1 for _ in i)]).sum()
  File "high_performance_pyspark/bad_pyspark.py", line 32, in <lambda>
    transform2 = transform1.map(lambda x: x / 0)
ZeroDivisionError: integer division or modulo by zero

    at org.apache.spark.api.python.PythonRunner$$anon$1
    .read(PythonRDD.scala:193)
    at org.apache.spark.api.python.PythonRunner$$anon$1
    .<init>(PythonRDD.scala:234)
    at org.apache.spark.api.python.PythonRunner.compute(PythonRDD.scala:152)
    at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:63)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:323)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:287)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:87)
    at org.apache.spark.scheduler.Task.run(Task.scala:113)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:313)
    at java.util.concurrent.ThreadPoolExecutor
    .runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker
    .run(ThreadPoolExecutor.java:617)
    ... 1 more
```

Logging is always important, and unlike JVM Spark applications we can't access the `log4j` logger to do our work for us.[10] For Python logging the simplest option is printing to `stdout`, which will end up in the `stderr` of the logs of the worker—however, this makes it difficult to tune logging levels. Instead, use a library like the standard `logging` library with stdout/stderr append functions.

For YARN users you can instead have the `logging` library write to a file under the `LOG_DIRS` environment variable, which will then be picked up as part of log aggregation later.

Debugging RDD skew in PySpark can become more challanging thanks to an often overlooked feature known as "batch serialization." This feature isn't normally given a lot of discussion as at large enough datasets, the impact of batch serializations tends to have minimal effects. This can quickly become confusing when following the standard debugging practice of sampling your data down to a small manageable set and trying to reproduce cluster behavior locally.

10 Technically you can access it on the driver program using `Py4J`, but the gateway isn't set up on workers, which is often where logging is most important.

Using Python with YARN may result in memory overhead errors that appear as out-of-memory errors. In the first part of this appendix we introduced memory overhead as some extra space required (see Figure A-1), but when running Python our entire Python process needs to fit inside of this "overhead" space. These can be difficult to debug as the error messages are the same for a few different situations.

The first possible cause of memory errors is unbalanced or otherwise large partitions. If the partitions are too large there may not be enough room for the Python workers to load the data. The web UI is one of the simpler places to check the partition sizes. If the partitions are unbalanced, a simple repartitioning can often do the trick, although the issues of key skew discussed in Chapter 6 can come into play.

The second possibility is simply not having enough overhead allocated for Python. The name "memory overhead" can be somewhat confusing but the Python worker is only able to use the "overhead" space left over in the container after the JVM has used the rest of the space. The default configuration of `spark.yarn.executor.memoryOverhead` is only 384 MB or 10% of the entire container (whichever is larger), which for PySpark users is often not a reasonable value. The related configuration variables for application master (AM) and driver (depending on your deployment mode) are `spark.yarn.am.memoryOverhead` and `spark.yarn.driver.memoryOverhead`.

Debugging conclusion

While debugging in Spark does indeed have some unique challenges, it's important to remember some of its many benefits for debugging. One of the strongest is that its local mode allows us to quickly create a "fake" cluster for testing or debugging without having to go through a long setup process. The other is that our jobs often run faster than traditional distributed systems, so we can often quickly experiment (either in local mode or our test cluster) to narrow down the source of the error. Echoing the conclusion in Chapter 10, may you need to apply the debugging section of this book as little as possible and may your adventures in Apache Spark be fun.

About the Authors

Holden Karau is transgender Canadian, and an active open source contributor. When not in San Francisco working as a software development engineer at IBM's Spark Technology Center, Holden talks internationally on Apache Spark and holds office hours at coffee shops at home and abroad. She is a Spark committer with frequent contributions, specializing in PySpark and Machine Learning. Prior to IBM she worked on a variety of distributed, search, and classification problems at Alpine, Databricks, Google, Foursquare, and Amazon. She graduated from the University of Waterloo with a Bachelor of Mathematics in Computer Science. Outside of software she enjoys playing with fire, welding, scooters, poutine, and dancing.

Rachel Warren is a data scientist and software engineer at Alpine Data Labs, where she uses Spark to address real-world data processing challenges. She has experience working as an analyst both in industry and academia. She graduated with a degree in Computer Science from Wesleyan University in Connecticut.

Colophon

The animal on the cover of *High Performance Spark* is a fire-tailed sunbird (*Aethopyga ignicauda*) native to Southeast Asia and the Indian subcontinent. Sunbirds are very distant relatives of the hummingbirds of the Americas and the honeyeaters of Australia.

As their name suggests, these birds (especially males) have very vivid coloring. Their tails and the back of their necks are red, their wings are green, their bellies are yellow and orange, and their heads are an iridescent blue. Male fire-tailed sunbirds are slightly larger than females, at an average of 15 centimeters long. Both members of a mating pair participate in feeding their chicks.

The fire-tailed sunbird's preferred habitat is conifer forest, where it consumes a diet of insects and nectar. The downward curve of its beak and a tubular tongue help it reach into flowers for food.

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

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