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 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.

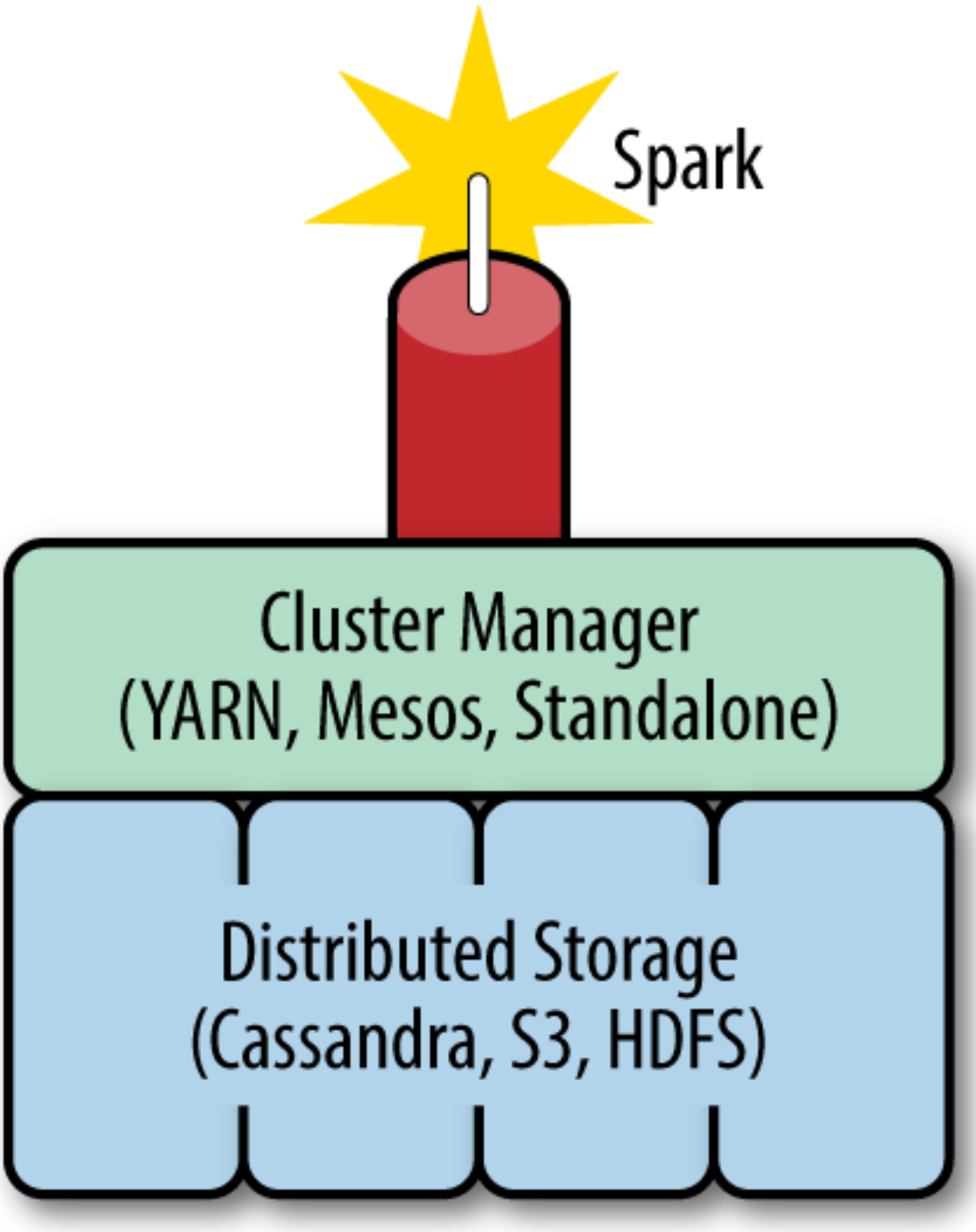

Figure 2-1. A diagram of the data processing ecosystem including Spark

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.

TIP

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 the window sizes used for batches. We offer some tips for using Spark Streaming in "Stream Processing with Spark".

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 "GraphX".

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 "Additional formats" and then in more detail in "Using Community Packages and Libraries".

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.

WARNING

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, availableon 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.

WARNING

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") 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 "Types of Reuse: Cache, Persist, Checkpoint, Shuffle Files". 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 "LRU Caching".

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 (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".

`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 "Using the Spark Partitioner Object".

`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.

TIP

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`.

WARNING

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" 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.

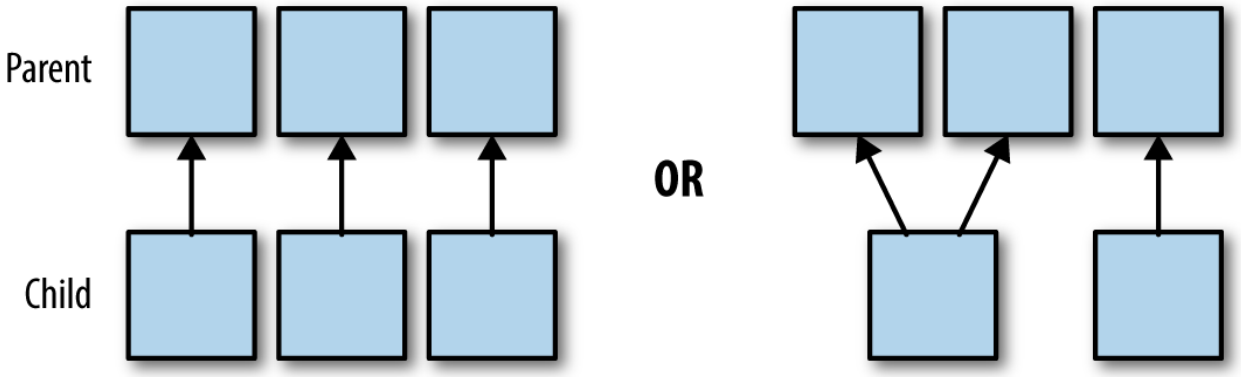

Figure 2-2. A simple diagram of dependencies between partitions for narrow transformations

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.

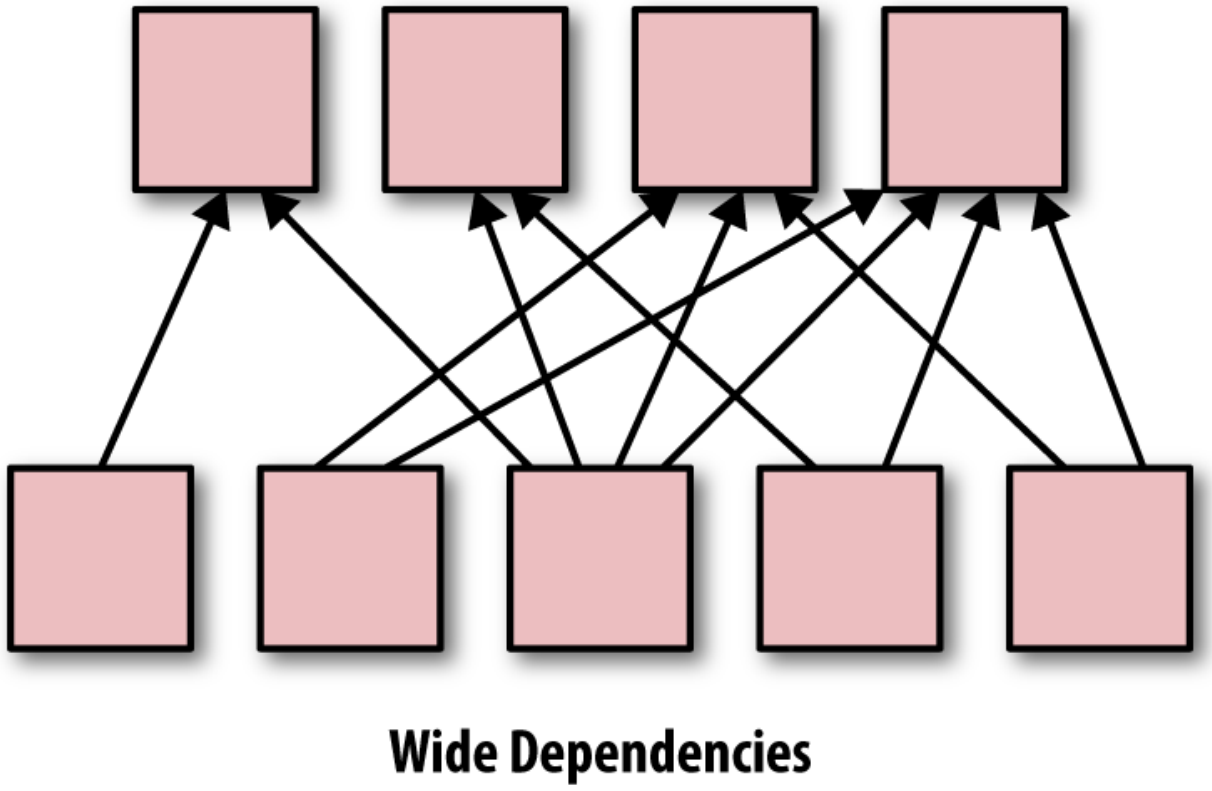

Figure 2-3. A simple diagram of dependencies between partitions for wide transformations

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 "Core Spark Joins".

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".

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`.

NOTE

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.

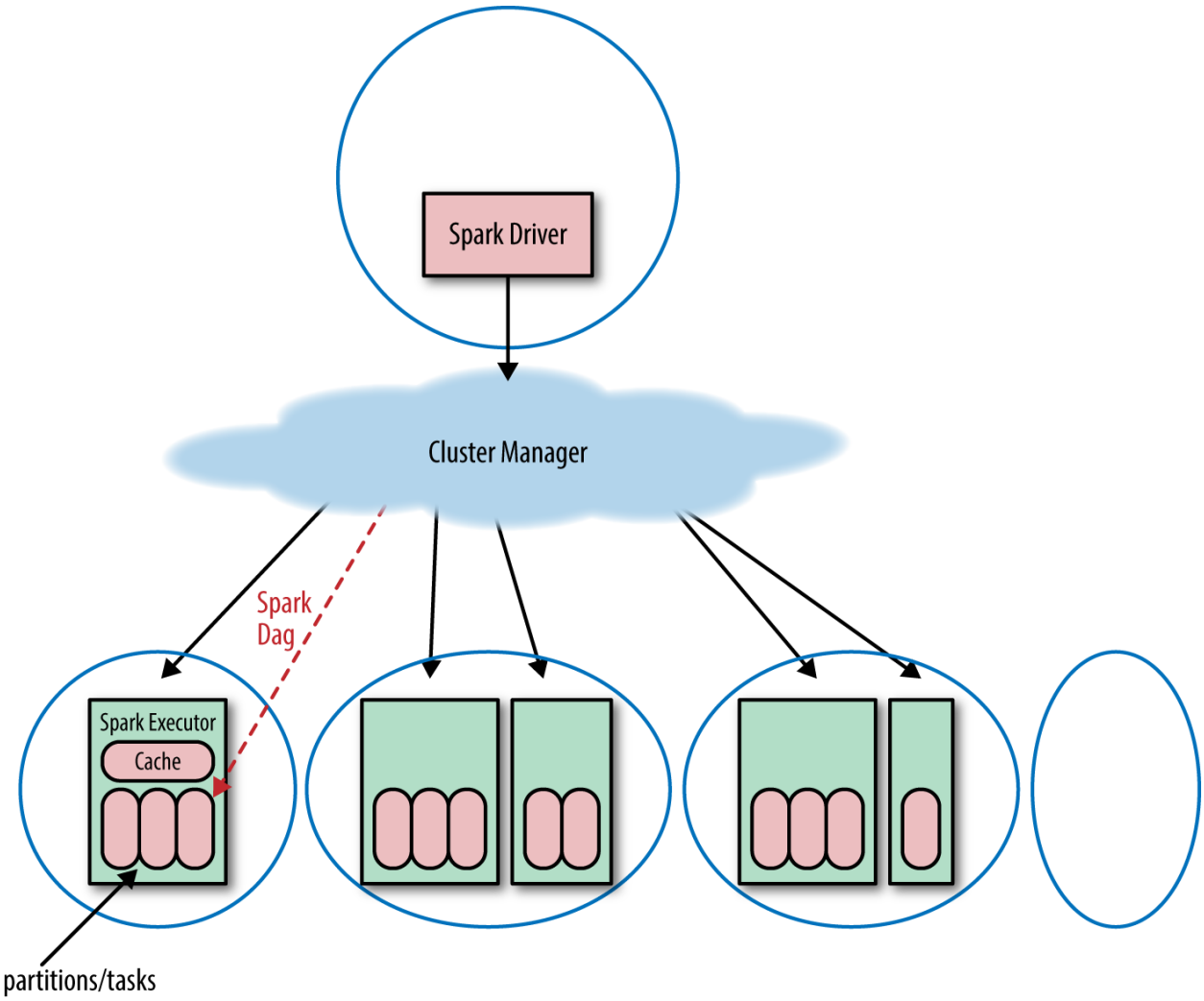

Figure 2-4. Starting a Spark application on a distributed system

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.

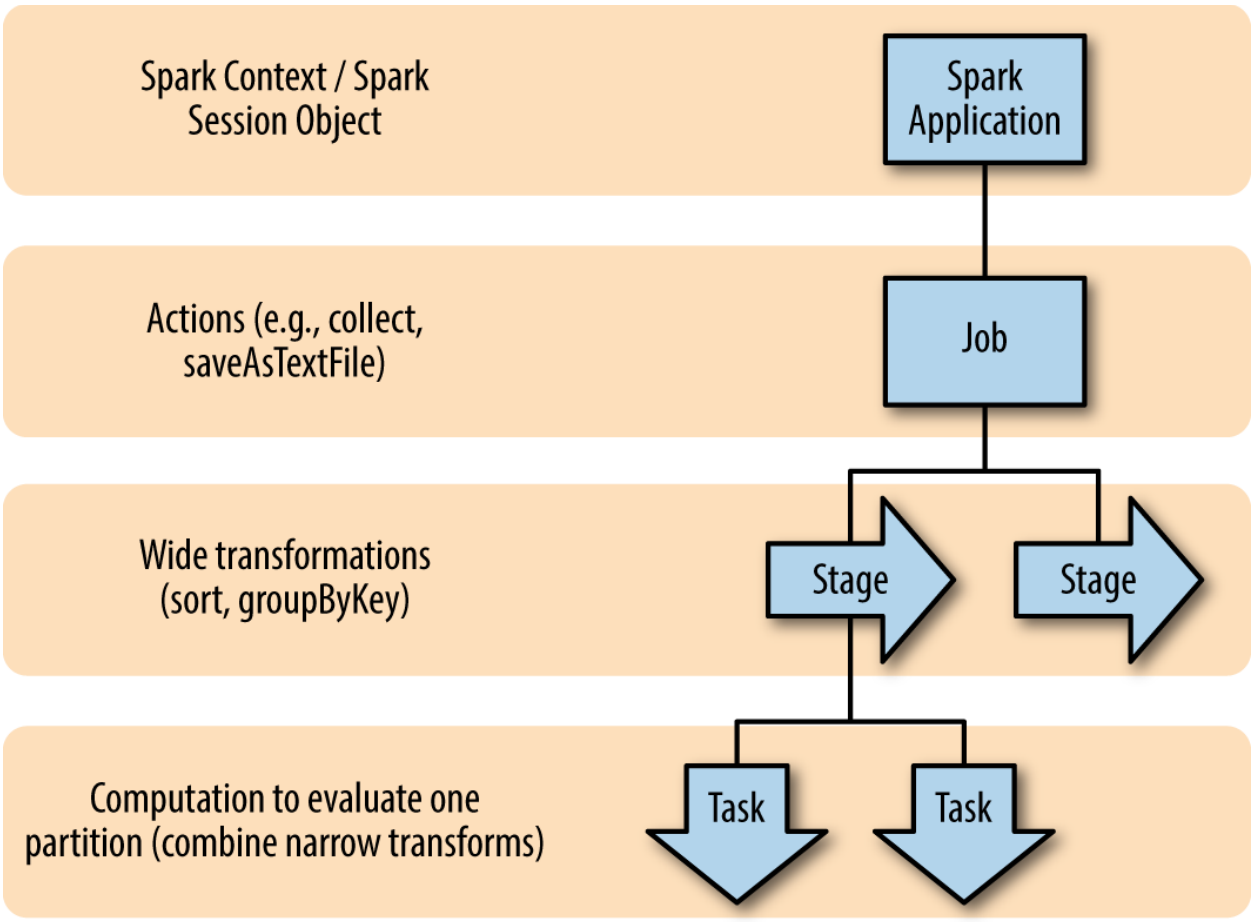

Figure 2-5. The Spark application tree

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", 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", 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.

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.

TIP

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.

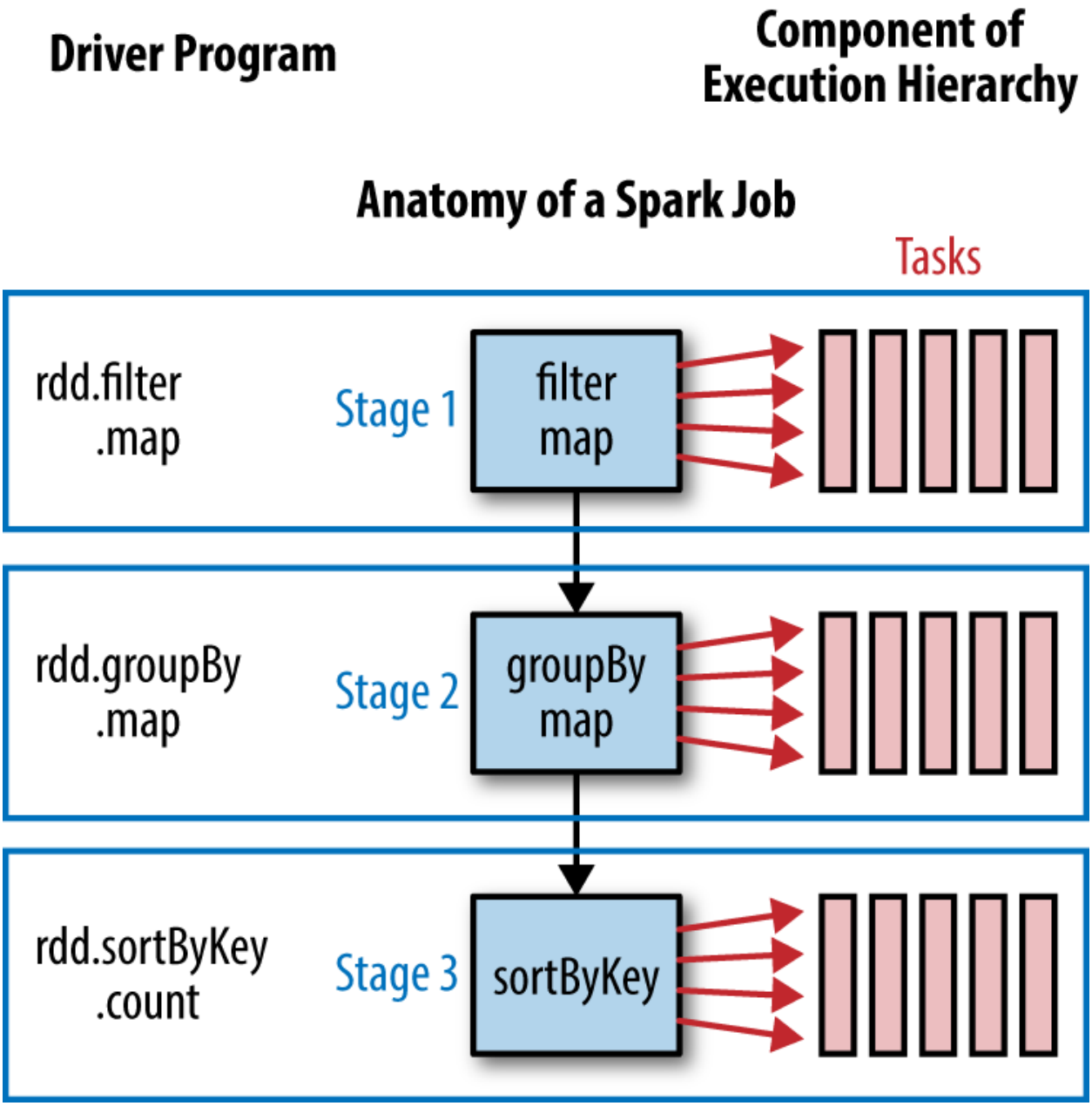

Figure 2-6. A stage diagram for the simple Spark program shown in Example 2-3

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").

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.

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.

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.

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

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

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