Spark IN ACTION

Petar Zečević
Marko Bonaći

SAMPLE CHAPTER

MANNING

Spark in Action

by Petar Zečević
and Marko Bonači

Chapter 1

brief contents

Part 1

First steps

We begin this book with an introduction to Apache Spark and its rich API. Understanding the information in part 1 is important for writing high-quality Spark programs and is an excellent foundation for the rest of the book.

Chapter 1 roughly describes Spark's main features and compares them with Hadoop's MapReduce and other tools from the Hadoop ecosystem. It also includes a description of the spark-in-action virtual machine we've prepared for you, which you can use to run the examples in the book.

Chapter 2 further explores the VM, teaches you how to use Spark's command-line interface (*spark-shell*), and uses several examples to explain *resilient distributed datasets* (RDDs)—the central abstraction in Spark.

In chapter 3, you'll learn how to set up Eclipse to write standalone Spark applications. Then you'll write such an application to analyze GitHub logs and execute the application by submitting it to a Spark cluster.

Chapter 4 explores the Spark core API in more detail. Specifically, it shows you how to work with key-value pairs and explains how data partitioning and *shuffling* work in Spark. It also teaches you how to group, sort, and join data, and how to use accumulators and broadcast variables.

1 Introduction to Apache Spark

This chapter covers

- What Spark brings to the table
- Spark components
- Spark program flow
- Spark ecosystem
- Downloading and starting the spark-in-action virtual machine

Apache Spark is usually defined as a fast, general-purpose, distributed computing platform. Yes, it sounds a bit like marketing speak at first glance, but we could hardly come up with a more appropriate label to put on the Spark box.

Apache Spark really did bring a revolution to the big data space. Spark makes efficient use of memory and can execute equivalent jobs 10 to 100 times faster than Hadoop's MapReduce. On top of that, Spark's creators managed to abstract away the fact that you're dealing with a cluster of machines, and instead present you with a set of collections-based APIs. Working with Spark's collections feels like working with local Scala, Java, or Python collections, but Spark's collections reference data distributed on many nodes. Operations on these collections get translated to complicated parallel programs without the user being necessarily aware of the fact, which is a truly powerful concept.

In this chapter, we first shed light on the main Spark features and compare Spark to its natural predecessor: Hadoop's MapReduce. Then we briefly explore Hadoop's ecosystem—a collection of tools and languages used together with Hadoop for big data operations—to see how Spark fits in. We give you a brief overview of Spark's components and show you how a typical Spark program executes using a simple "Hello World" example. Finally, we help you download and set up the spark-in-action virtual machine we prepared for running the examples in the book.

We've done our best to write a comprehensive guide to Spark architecture, its components, its runtime environment, and its API, while providing concrete examples and real-life case studies. By reading this book and, more important, by sifting through the examples, you'll gain the knowledge and skills necessary for writing your own high-quality Spark programs and managing Spark applications.

1.1 What is Spark?

Apache Spark is an exciting new technology that is rapidly superseding Hadoop's MapReduce as the preferred big data processing platform. Hadoop is an open source, distributed, Java computation framework consisting of the Hadoop Distributed File System (HDFS) and MapReduce, its execution engine. Spark is similar to Hadoop in that it's a distributed, general-purpose computing platform. But Spark's unique design, which allows for keeping large amounts of data in memory, offers tremendous performance improvements. Spark programs can be 100 times faster than their MapReduce counterparts.

Spark was originally conceived at Berkeley's AMPLab by Matei Zaharia, who went on to cofound Databricks, together with his mentor Ion Stoica, as well as Reynold Xin, Patrick Wendell, Andy Konwinski, and Ali Ghodsi. Although Spark is open source, Databricks is the main force behind Apache Spark, contributing more than 75% of Spark's code. It also offers Databricks Cloud, a commercial product for big data analysis based on Apache Spark.

By using Spark's elegant API and runtime architecture, you can write distributed programs in a manner similar to writing local ones. Spark's collections abstract away the fact that they're potentially referencing data distributed on a large number of nodes. Spark also allows you to use functional programming methods, which are a great match for data-processing tasks.

By supporting Python, Java, Scala, and, most recently, R, Spark is open to a wide range of users: to the science community that traditionally favors Python and R, to the still-widespread Java community, and to people using the increasingly popular Scala, which offers functional programming on the Java virtual machine (JVM).

Finally, Spark combines MapReduce-like capabilities for batch programming, real-time data-processing functions, SQL-like handling of structured data, graph algorithms, and machine learning, all in a single framework. This makes it a one-stop shop for most of your big data-crunching needs. It's no wonder, then, that Spark is one of the busiest and fastest-growing Apache Software Foundation projects today.

But some applications aren't appropriate for Spark. Because of its distributed architecture, Spark necessarily brings some overhead to the processing time. This overhead is negligible when handling large amounts of data; but if you have a dataset that can be handled by a single machine (which is becoming ever more likely these days), it may be more efficient to use some other framework optimized for that kind of computation. Also, Spark wasn't made with online transaction processing (OLTP) applications in mind (fast, numerous, atomic transactions). It's better suited for online analytical processing (OLAP): batch jobs and data mining.

1.1.1 The Spark revolution

Although the last decade saw Hadoop's wide adoption, Hadoop is not without its shortcomings. It's powerful, but it can be slow. This has opened the way for newer technologies, such as Spark, to solve the same challenges Hadoop solves, but more efficiently. In the next few pages, we'll discuss Hadoop's shortcomings and how Spark answers those issues.

The Hadoop framework, with its HDFS and MapReduce data-processing engine, was the first that brought distributed computing to the masses. Hadoop solved the three main problems facing any distributed data-processing endeavor:

- *Parallelization*—How to perform subsets of the computation simultaneously
- *Distribution*—How to distribute the data
- *Fault tolerance*—How to handle component failure

NOTE Appendix A describes MapReduce in more detail.

On top of that, Hadoop clusters are often made of commodity hardware, which makes Hadoop easy to set up. That's why the last decade saw its wide adoption.

1.1.2 MapReduce's shortcomings

Although Hadoop is the foundation of today's big data revolution and is actively used and maintained, it still has its shortcomings, and they mostly pertain to its MapReduce component. MapReduce job results need to be stored in HDFS before they can be used by another job. For this reason, MapReduce is inherently bad with iterative algorithms.

Furthermore, many kinds of problems don't easily fit MapReduce's two-step paradigm, and decomposing every problem into a series of these two operations can be difficult. The API can be cumbersome at times.

Hadoop is a rather low-level framework, so myriad tools have sprung up around it: tools for importing and exporting data, higher-level languages and frameworks for manipulating data, tools for real-time processing, and so on. They all bring additional complexity and requirements with them, which complicates any environment. Spark solves many of these issues.

1.1.3 What Spark brings to the table

Spark's core concept is an in-memory execution model that enables caching job data in memory instead of fetching it from disk every time, as MapReduce does. This can speed the execution of jobs up to 100 times,[1] compared to the same jobs in MapReduce; it has the biggest effect on iterative algorithms such as machine learning, graph algorithms, and other types of workloads that need to reuse data.

Imagine you have city map data stored as a graph. The vertices of this graph represent points of interest on the map, and the edges represent possible routes between them, with associated distances. Now suppose you need to find a spot for a new ambulance station that will be situated as close as possible to all the points on the map. That spot would be the center of your graph. It can be found by first calculating the shortest path between all the vertices and then finding the *farthest point distance* (the maximum distance to any other vertex) for each vertex, and finally finding the vertex with the smallest farthest point distance. Completing the first phase of the algorithm, finding the shortest path between all vertices, in a parallel manner is the most challenging (and complicated) part, but it's not impossible.[2]

In the case of MapReduce, you'd need to store the results of each of these three phases on disk (HDFS). Each subsequent phase would read the results of the previous one from disk. But with Spark, you can find the shortest path between all vertices and cache that data in memory. The next phase can use that data from memory, find the farthest point distance for each vertex, and cache its results. The last phase can go through this final cached data and find the vertex with the minimum farthest point distance. You can imagine the performance gains compared to reading and writing to disk every time.

Spark performance is so good that in October 2014 it won the Daytona Gray Sort contest and set a world record (jointly with TritonSort, to be fair) by sorting 100 TB in 1,406 seconds (see http://sortbenchmark.org).

SPARK'S EASE OF USE

The Spark API is much easier to use than the classic MapReduce API. To implement the classic word-count example from appendix A as a MapReduce job, you'd need three classes: the main class that sets up the job, a `Mapper`, and a `Reducer`, each 10 lines long, give or take a few.

---

[1] See "Shark: SQL and Rich Analytics at Scale" by Reynold Xin et al., http://mng.bz/gFry.

[2] See "A Scalable Parallelization of All-Pairs Shortest Path Algorithm for a High Performance Cluster Environment" by T. Srinivasan et al., http://mng.bz/5TMT.

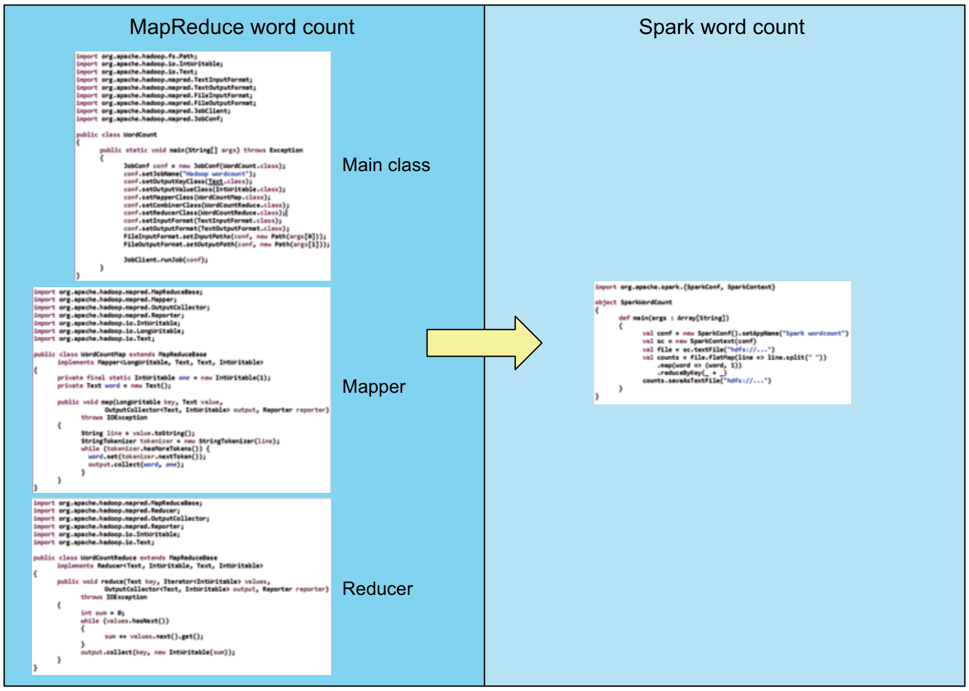
By contrast, the following is all it takes for the same Spark program written in Scala:

```
val spark = SparkSession.builder().appName("Spark wordcount")
val file = spark.sparkContext.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
    .map(word => (word, 1)).countByKey()
counts.saveAsTextFile("hdfs://...")
```

Figure 1.1. shows this graphically.

Spark supports the Scala, Java, Python, and R programming languages, so it's accessible to a much wider audience. Although Java is supported, Spark can take advantage of Scala's versatility, flexibility, and functional programming concepts, which are a much better fit for data analysis. Python and R are widespread among data scientists and in the scientific community, which brings those users on par with Java and Scala developers.

Furthermore, the Spark shell (read-eval-print loop [REPL]) offers an interactive console that can be used for experimentation and idea testing. There's no need for compilation and deployment just to find out something isn't working (again). REPL can even be used for launching jobs on the full set of data.



```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.TextOutputFormat;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;

public class WordCount
{
	public static void main(String[] args) throws Exception
	{
		JobConf conf = new JobConf(WordCount.class);
		conf.setJobName("Hadoop wordcount");
		conf.setOutputKeyClass(Text.class);
		conf.setOutputValueClass(IntWritable.class);
		conf.setMapperClass(WordCountMap.class);
		conf.setCombinerClass(WordCountReduce.class);
		conf.setReducerClass(WordCountReduce.class);
		conf.setInputFormat(TextInputFormat.class);
		conf.setOutputFormat(TextOutputFormat.class);
		FileInputFormat.setInputPaths(conf, new Path(args[0]));
		FileOutputFormat.setOutputPath(conf, new Path(args[1]));

		JobClient.runJob(conf);
	}
}
```

```
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;

public class WordCountMap extends MapReduceBase
	implements Mapper<LongWritable, Text, Text, IntWritable>
{
	private final static IntWritable one = new IntWritable(1);
	private Text word = new Text();

	public void map(LongWritable key, Text value,
			OutputCollector<Text, IntWritable> output, Reporter reporter)
		throws IOException
	{
		String line = value.toString();
		StringTokenizer tokenizer = new StringTokenizer(line);
		while (tokenizer.hasMoreTokens()) {
		  word.set(tokenizer.nextToken());
		  output.collect(word, one);
		}
	}
}
```

```
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;

public class WordCountReduce extends MapReduceBase
	implements Reducer<Text, IntWritable, Text, IntWritable>
{
	public void reduce(Text key, Iterator<IntWritable> values,
			OutputCollector<Text, IntWritable> output, Reporter reporter)
		throws IOException
	{
		int sum = 0;
		while (values.hasNext())
		{
			sum += values.next().get();
		}
		output.collect(key, new IntWritable(sum));
	}
}
```

```
import org.apache.spark.{SparkConf, SparkContext}

object SparkWordCount
{
	def main(args : Array[String])
	{
		val conf = new SparkConf().setAppName("Spark wordcount")
		val sc = new SparkContext(conf)
		val file = sc.textFile("hdfs://...")
		val counts = file.flatMap(line => line.split(" "))
			.map(word => (word, 1))
			.reduceByKey(_ + _)
		counts.saveAsTextFile("hdfs://...")
	}
}
```

Figure 1.1 A word-count program demonstrates Spark's conciseness and simplicity. The program is shown implemented in Hadoop's MapReduce framework on the left and as a Spark Scala program on the right.

Finally, Spark can run on several types of clusters: Spark standalone cluster, Hadoop's YARN (yet another resource negotiator), and Mesos. This gives it additional flexibility and makes it accessible to a larger community of users.

SPARK AS A UNIFYING PLATFORM

An important aspect of Spark is its combination of the many functionalities of the tools in the Hadoop ecosystem into a single unifying platform. The execution model is general enough that the single framework can be used for stream data processing, machine learning, SQL-like operations, and graph and batch processing. Many roles can work together on the same platform, which helps bridge the gap between programmers, data engineers, and data scientists. And the list of functions that Spark provides is continuing to grow.

SPARK ANTI-PATTERNS

Spark isn't suitable, though, for asynchronous updates to shared data[3] (such as online transaction processing, for example), because it has been created with batch analytics in mind. (Spark streaming is simply batch analytics applied to data in a time window.) Tools specialized for those use cases will still be necessary.

Also, if you don't have a large amount of data, Spark may not be required, because it needs to spend some time setting up jobs, tasks, and so on. Sometimes a simple relational database or a set of clever scripts can be used to process data more quickly than a distributed system such as Spark. But data has a tendency to grow, and it may outgrow your relational database management system (RDBMS) or your clever scripts rather quickly.

1.2 Spark components

Spark consists of several purpose-built components. These are Spark Core, Spark SQL, Spark Streaming, Spark GraphX, and Spark MLlib, as shown in figure 1.2.

These components make Spark a feature-packed *unifying platform*: it can be used for many tasks that previously had to be accomplished with several different frameworks. A brief description of each Spark component follows.

1.2.1 Spark Core

Spark Core contains basic Spark functionalities required for running jobs and needed by other components. The most important of these is the *resilient distributed dataset* (RDD),[4] which is the main element of the Spark API. It's an abstraction of a *distributed* collection of items with operations and transformations applicable to the dataset. It's *resilient* because it's capable of rebuilding datasets in case of node failures.

[3] See "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing" by Matei Zaharia et al., http://mng.bz/57uJ.

[4] RDDs are explained in chapter 2. Because they're the fundamental abstraction of Spark, they're also covered in detail in chapter 4.

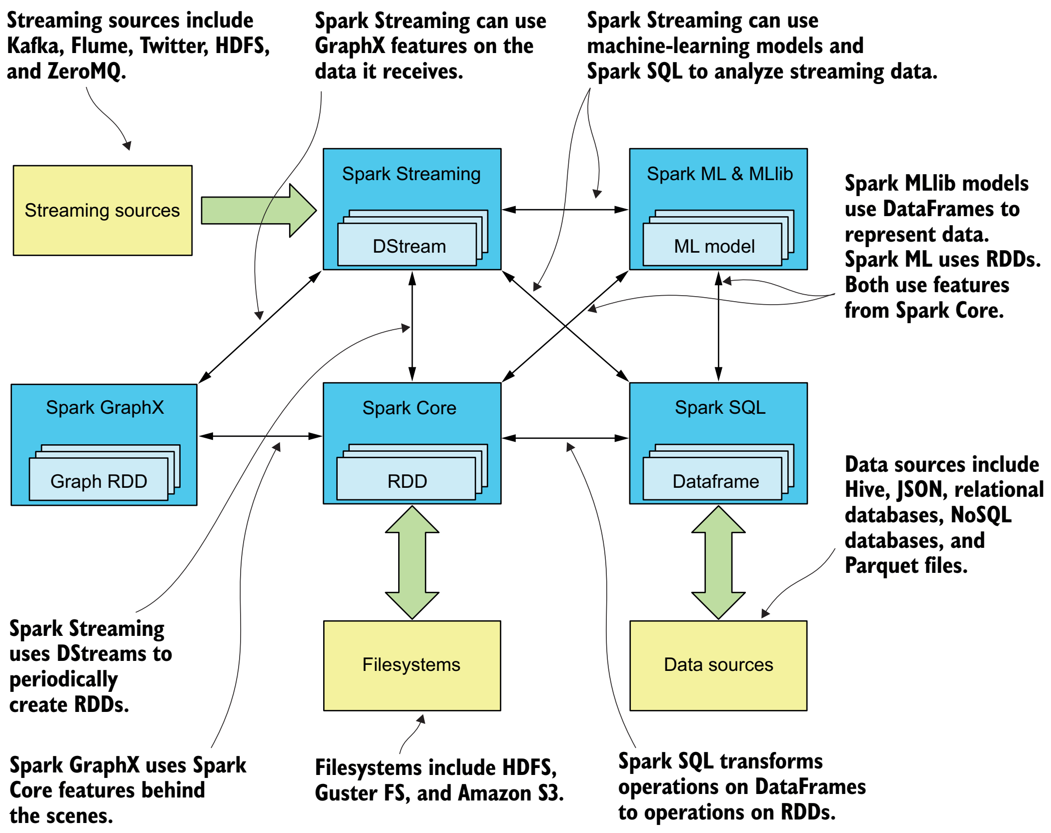

Streaming sources include Kafka, Flume, Twitter, HDFS, and ZeroMQ.
Spark Streaming can use GraphX features on the data it receives.
Spark Streaming can use machine-learning models and Spark SQL to analyze streaming data.
Streaming sources
Spark Streaming
DStream
Spark ML & MLlib
ML model
Spark MLlib models use DataFrames to represent data. Spark ML uses RDDs. Both use features from Spark Core.
Spark GraphX
Graph RDD
Spark Core
RDD
Spark SQL
Dataframe
Data sources include Hive, JSON, relational databases, NoSQL databases, and Parquet files.
Spark Streaming uses DStreams to periodically create RDDs.
Filesystems
Data sources
Spark GraphX uses Spark Core features behind the scenes.
Filesystems include HDFS, Guster FS, and Amazon S3.
Spark SQL transforms operations on DataFrames to operations on RDDs.

Figure 1.2 Main Spark components and various runtime interactions and storage options

Spark Core contains logic for accessing various filesystems, such as HDFS, GlusterFS, Amazon S3, and so on. It also provides a means of information sharing between computing nodes with broadcast variables and accumulators. Other fundamental functions, such as networking, security, scheduling, and data shuffling, are also part of Spark Core.

1.2.2 Spark SQL

Spark SQL provides functions for manipulating large sets of distributed, structured data using an SQL subset supported by Spark and Hive SQL (HiveQL). With `DataFrames` introduced in Spark 1.3, and `DataSets` introduced in Spark 1.6, which simplified handling of structured data and enabled radical performance optimizations, Spark SQL became one of the most important Spark components. Spark SQL can also be used for reading and writing data to and from various structured formats and data sources, such as JavaScript Object Notation (JSON) files, Parquet files (an increasingly popular file format that allows for storing a schema along with the data), relational databases, Hive, and others.

Operations on DataFrames and DataSets at some point translate to operations on RDDs and execute as ordinary Spark jobs. Spark SQL provides a query optimization framework called Catalyst that can be extended by custom optimization rules. Spark SQL also includes a Thrift server, which can be used by external systems, such as business intelligence tools, to query data through Spark SQL using classic JDBC and ODBC protocols.

1.2.3 Spark Streaming

Spark Streaming is a framework for ingesting real-time streaming data from various sources. The supported streaming sources include HDFS, Kafka, Flume, Twitter, ZeroMQ, and custom ones. Spark Streaming operations recover from failure automatically, which is important for online data processing. Spark Streaming represents streaming data using *discretized streams* (DStreams), which periodically create RDDs containing the data that came in during the last time window.

Spark Streaming can be combined with other Spark components in a single program, unifying real-time processing with machine learning, SQL, and graph operations. This is something unique in the Hadoop ecosystem. And since Spark 2.0, the new Structured Streaming API makes Spark streaming programs more similar to Spark batch programs.

1.2.4 Spark MLlib

Spark MLlib is a library of machine-learning algorithms grown from the MLbase project at UC Berkeley. Supported algorithms include logistic regression, naïve Bayes classification, support vector machines (SVMs), decision trees, random forests, linear regression, and k-means clustering.

Apache Mahout is an existing open source project offering implementations of distributed machine-learning algorithms running on Hadoop. Although Apache Mahout is more mature, both Spark MLlib and Mahout include a similar set of machine-learning algorithms. But with Mahout migrating from MapReduce to Spark, they're bound to be merged in the future.

Spark MLlib handles machine-learning models used for transforming datasets, which are represented as RDDs or DataFrames.

1.2.5 Spark GraphX

Graphs are data structures comprising vertices and the edges connecting them. GraphX provides functions for building graphs, represented as *graph RDDs*: EdgeRDD and VertexRDD. GraphX contains implementations of the most important algorithms of graph theory, such as page rank, connected components, shortest paths, SVD++, and others. It also provides the Pregel message-passing API, the same API for large-scale graph processing implemented by Apache Giraph, a project with implementations of graph algorithms and running on Hadoop.

1.3 Spark program flow

Let's see what a typical Spark program looks like. Imagine that a 300 MB log file is stored in a three-node HDFS cluster. HDFS automatically splits the file into 128 MB

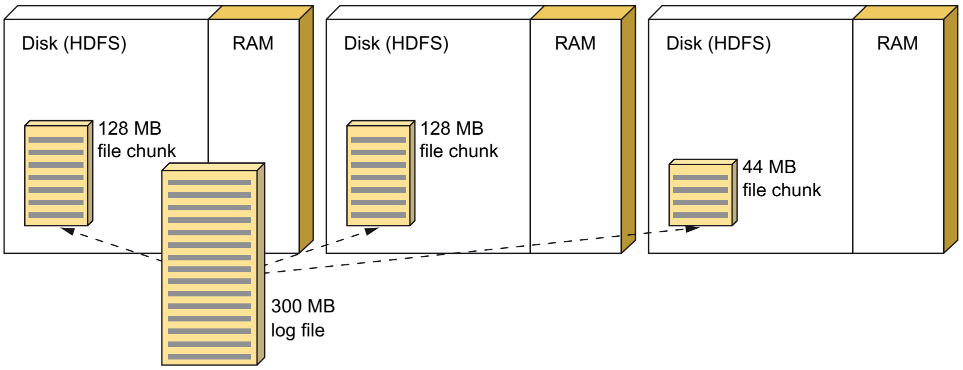


Figure 1.3 Storing a 300 MB log file in a three-node Hadoop cluster

parts (*blocks*, in Hadoop terminology) and places each part on a separate node of the cluster[5] (see figure 1.3). Let's assume Spark is running on YARN, inside the same Hadoop cluster.

A Spark data engineer is given the task of analyzing how many errors of type `OutOfMemoryError` have happened during the last two weeks. Mary, the engineer, knows that the log file contains the last two weeks of logs of the company's application server cluster. She sits at her laptop and starts to work.

She first starts her *Spark shell* and establishes a connection to the Spark cluster. Next, she loads the log file from HDFS (see figure 1.4) by using this (Scala) line:

```
val lines = sc.textFile("hdfs://path/to/the/file")
```

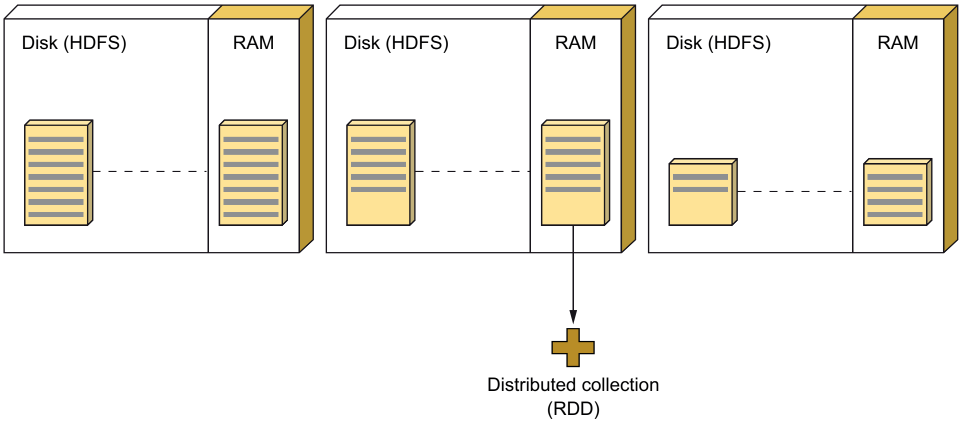


Figure 1.4 Loading a text file from HDFS

[5] Although it's not relevant to our example, we should probably mention that HDFS replicates each block to two additional nodes (if the default replication factor of 3 is in effect).

To achieve maximum *data locality*,[6] the loading operation asks Hadoop for the locations of each block of the log file and then transfers all the blocks into RAM of the cluster's nodes. Now Spark has a reference to each of those blocks (*partitions*, in Spark terminology) in RAM. The sum of those partitions is a distributed collection of lines from the log file referenced by an RDD. Simplifying, we can say that RDDs allow you to work with a distributed collection the same way you would work with any local, nondistributed one. You don't have to worry about the fact that the collection is distributed, nor do you have to handle node failures yourself.

In addition to automatic fault tolerance and distribution, the RDD provides an elaborate API, which allows you to work with a collection in a functional style. You can filter the collection; map over it with a function; reduce it to a cumulative value; subtract, intersect, or create a union with another RDD, and so on.

Mary now has a reference to the RDD, so in order to find the error count, she first wants to remove all the lines that don't have an `OutOfMemoryError` substring. This is a job for the `filter` function, which she calls like this:

```
val oomLines = lines.filter(l => l.contains("OutOfMemoryError")).cache()
```

After filtering the collection so it contains the subset of data that she needs to analyze (see figure 1.5), Mary calls `cache` on it, which tells Spark to leave that RDD in memory across jobs. Caching is the basic component of Spark's performance improvements we mentioned before. The benefits of caching the RDD will become apparent later.

Now she is left with only those lines that contain the error substring. For this simple example, we'll ignore the possibility that the `OutOfMemoryError` string might occur in multiple lines of a single error. Our data engineer counts the remaining lines

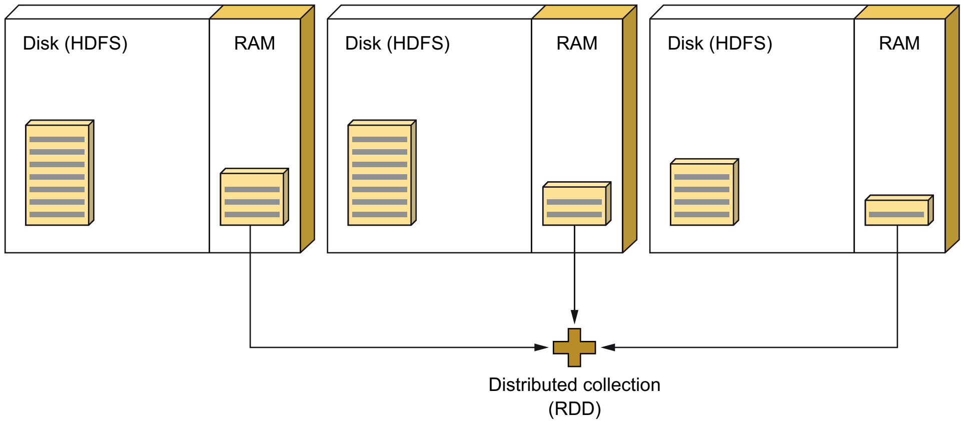


Figure 1.5 Filtering the collection to contain only lines containing the `OutOfMemoryError` string

[6] Data locality is honored if each block gets loaded in the RAM of the same node where it resides in HDFS. The whole point is to try to avoid having to transfer large amounts of data over the wire.

and reports the result as the number of out-of-memory errors that occurred in the last two weeks:

```
val result = oomLines.count()
```

Spark enabled her to perform distributed filtering and counting of the data with only three lines of code. Her little program was executed on all three nodes in parallel.

If she now wants to further analyze lines with `OutOfMemoryErrors`, and perhaps call `filter` again (but with other criteria) on an `oomLines` object that was previously cached in memory, Spark won't load the file from HDFS again, as it would normally do. Spark will load it from the cache.

1.4 Spark ecosystem

We've already mentioned the Hadoop ecosystem, consisting of interface, analytic, cluster-management, and infrastructure tools. Some of the most important ones are shown in figure 1.6.

Figure 1.6 is by no means complete.[7] You could argue that we failed to add one tool or another, but a complete list of tools would be hard to fit in this section. We believe, though, that this list represents a good subset of the most prominent tools in the Hadoop ecosystem.

If you compare the functionalities of Spark components with the tools in the Hadoop ecosystem, you can see that some of the tools are suddenly superfluous. For example, Apache Giraph can be replaced by Spark GraphX, and Spark MLlib can be

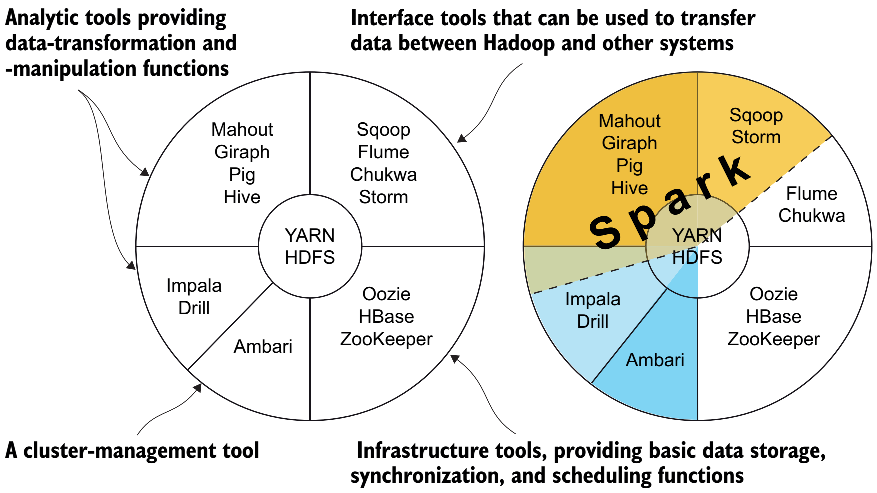

Analytic tools providing data-transformation and -manipulation functions
Interface tools that can be used to transfer data between Hadoop and other systems
Mahout
Giraph
Pig
Hive
Sqoop
Flume
Chukwa
Storm
YARN
HDFS
Impala
Drill
Ambari
Oozie
HBase
ZooKeeper
Mahout
Giraph
Pig
Hive
Sqoop
Storm
Spark
Flume
Chukwa
YARN
HDFS
Impala
Drill
Ambari
Oozie
HBase
ZooKeeper
A cluster-management tool
Infrastructure tools, providing basic data storage, synchronization, and scheduling functions

Figure 1.6 Basic infrastructure, interface, analytic, and management tools in the Hadoop ecosystem, with some of the functionalities that Spark incorporates or makes obsolete

[7] If you're interested, you can find a (hopefully) complete list of Hadoop-related tools and frameworks at http://hadoopecosystemtable.github.io.

used instead of Apache Mahout. Apache Storm's capabilities overlap greatly with those of Spark Streaming, so in many cases Spark Streaming can be used instead.

Apache Pig and Apache Sqoop aren't needed any longer, because the same functionalities are covered by Spark Core and Spark SQL. But even if you have legacy Pig workflows and need to run Pig, the Spork project enables you to run Pig on Spark.

Spark has no means of replacing the infrastructure and management of the Hadoop ecosystem tools (Oozie, HBase, and ZooKeeper), though. Oozie is used for scheduling different types of Hadoop jobs and now even has an extension for scheduling Spark jobs. HBase is a distributed and scalable database, which is something Spark doesn't provide. ZooKeeper provides fast and robust implementation of common functionalities many distributed applications need, like coordination, distributed synchronization, naming, and provisioning of group services. It is used for these purposes in many other distributed systems, too.

Impala and Drill can coexist alongside Spark, especially with Drill's coming support for Spark as an execution engine. But they're more like competing frameworks, mostly spanning the features of Spark Core and Spark SQL, which makes Spark feature-richer (pun not intended).

We said earlier that Spark doesn't need to use HDFS storage. In addition to HDFS, Spark can operate on data stored in Amazon S3 buckets and plain files. More exciting, it can also use Alluxio (formerly Tachyon), which is a memory-centric distributed filesystem, or other distributed filesystems, such as GlusterFS.

Another interesting fact is that Spark doesn't have to run on YARN. Apache Mesos and the Spark standalone cluster are alternative cluster managers for Spark. Apache Mesos is an advanced distributed systems kernel bringing distributed resource abstractions. It can scale to tens of thousands of nodes with full fault tolerance (we'll visit it in chapter 12). Spark Standalone is a Spark-specific cluster manager that is used in production today on multiple sites.

So if we switch from MapReduce to Spark and get rid of YARN and all the tools that Spark makes obsolete, what's left of the Hadoop ecosystem? To put it another way: Are we slowly moving toward a new big data standard: a *Spark ecosystem*?

1.5 Setting up the spark-in-action VM

In order to make it easy for you to set up a Spark learning environment, we prepared a virtual machine (VM) that you'll be using throughout this book. It will allow you to run all the examples from the book without surprises due to different versions of Java, Spark, or your OS. For example, you could have problems running the Spark examples on Windows; after all, Spark is developed on OS X and Linux, so, understandably, Windows isn't exactly in the focus. The VM will guarantee we're all on the same page, so to speak.

The VM consists of the following software stack:

- *64-bit Ubuntu OS, 14.04.4 (nicknamed Trusty)*—Currently the latest version with long-term support (LTS).

- *Java 8 (OpenJDK)*—Even if you plan on only using Spark from Python, you have to install Java, because Spark's Python API communicates with Spark running in a JVM.
- *Hadoop 2.7.2*—Hadoop isn't a hard requirement for using Spark. You can save and load files from your local filesystem, if you're running a local cluster, which is the case with our VM. But as soon as you set up a truly distributed Spark cluster, you'll need a distributed filesystem, such as Hadoop's HDFS. Hadoop installation will also come in handy in chapter 12 for trying out the methods of running Spark on YARN, Hadoop's execution environment.
- *Spark 2.0*—We included the latest Spark version at the time this book was finished. You can easily upgrade the Spark version in the VM, if you wish to do so, by following the instructions in chapter 2.
- *Kafka 0.8.2*—Kafka is a distributed messaging system, used in chapters 6 and 13.

We chose Ubuntu because it's a popular Linux distribution and Linux is the preferred Spark platform. If you've never worked with Ubuntu before, this could be your chance to start. We'll guide you, explaining commands and concepts as you progress through the chapters.

Here we'll explain only the basics: how to download, start, and stop the VM. We'll go into more details about using it in the next chapter.

1.5.1 Downloading and starting the virtual machine

To run the VM, you'll need a 64-bit OS with at least 3 GB of free memory and 15 GB of free disk space. You first need to install these two software packages for your platform:

- *Oracle VirtualBox*—Oracle's free, open source hardware virtualization software (www.virtualbox.org)
- *Vagrant*—HashiCorp's software for configuring portable development environments (www.vagrantup.com/downloads.html)

When you have these two installed, create a folder for hosting the VM (called, for example, spark-in-action), and enter it. Then download the Vagrant box metadata JSON file from our online repository. You can download it manually or use the `wget` command on Linux or Mac:

```
$ wget https://raw.githubusercontent.com/spark-in-action/first-edition/
➥ master/spark-in-action-box.json
```

Then issue the following command to download the VM itself:

```
$ vagrant box add spark-in-action-box.json
```

The Vagrant box metadata JSON file points to the Vagrant box file. The command will download the 5 GB VM box (this will probably take some time) and register it as the manning/spark-in-action Vagrant box. To use it, initialize the Vagrant VM in the current directory by issuing this command:

```
$ vagrant init manning/spark-in-action
```

Finally, start the VM with the `vagrant up` command (this will also allocate approximately 10 GB of disk space):

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Checking if box 'manning/spark-in-action' is up to date...
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
...
```

If you have several network interfaces on your machine, you'll be asked to choose one of them for connecting it to the VM. Choose the one with an access to the internet. For example:

```
==> default: Available bridged network interfaces:
1) 1x1 11b/g/n Wireless LAN PCI Express Half Mini Card Adapter
2) Cisco Systems VPN Adapter for 64-bit Windows
==> default: When choosing an interface, it is usually the one that is
==> default: being used to connect to the internet.
    default: Which interface should the network bridge to? 1
==> default: Preparing network interfaces based on configuration...
...
```

1.5.2 Stopping the virtual machine

You'll learn how to use the VM in the next chapter. For now, we'll only show you how to stop it. To power off the VM, issue the following command:

```
$ vagrant halt
```

This will stop the machine but preserve your work. If you wish to completely remove the VM and free up its space, you need to *destroy* it:

```
$ vagrant destroy
```

You can also remove the downloaded Vagrant box, which was used to create the VM, with this command:

```
$ vagrant box remove manning/spark-in-action
```

But we hope you won't feel the need for that for quite some time.

1.6 Summary

- Apache Spark is an exciting new technology that is rapidly superseding Hadoop's MapReduce as the preferred big data processing platform.
- Spark programs can be 100 times faster than their MapReduce counterparts.
- Spark supports the Java, Scala, Python, and R languages.
- Writing distributed programs with Spark is similar to writing local Java, Scala, or Python programs.

- Spark provides a unifying platform for batch programming, real-time data-processing functions, SQL-like handling of structured data, graph algorithms, and machine learning, all in a single framework.
- Spark isn't appropriate for small datasets, nor should you use it for OLTP applications.
- The main Spark components are Spark Core, Spark SQL, Spark Streaming, Spark MLlib, and Spark GraphX.
- RDDs are Spark's abstraction of distributed collections.
- Spark supersedes some of the tools in the Hadoop ecosystem.
- You'll use the spark-in-action VM to run the examples in this book.

DATA/PROGRAMMING

Spark IN ACTION

Zečević • Bonaći

Big data systems distribute datasets across clusters of machines, making it a challenge to efficiently query, stream, and interpret them. Spark can help. It is a processing system designed specifically for distributed data. It provides easy-to-use interfaces, along with the performance you need for production-quality analytics and machine learning. And Spark 2 adds improved programming APIs, better performance, and countless other upgrades.

Spark in Action teaches you the theory and skills you need to effectively handle batch and streaming data using Spark. You'll get comfortable with the Spark CLI as you work through a few introductory examples. Then, you'll start programming Spark using its core APIs. Along the way, you'll work with structured data using Spark SQL, process near-real-time streaming data, apply machine learning algorithms, and munge graph data using Spark GraphX. For a zero-effort startup, you can download the preconfigured virtual machine ready for you to try the book's code.

What's Inside

- Updated for Spark 2.0
- Real-life case studies
- Spark DevOps with Docker
- Examples in Scala, and online in Java and Python

Written for experienced programmers with some background in big data or machine learning.

Petar Zečević and **Marko Bonaći** are seasoned developers heavily involved in the Spark community.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/spark-in-action

Free eBook
SEE INSERT

“Dig in and get your hands dirty with one of the hottest data processing engines today. A great guide.”
—Jonathan Sharley
Pandora Media

“Must-have! Speed up your learning of Spark as a distributed computing framework.”
—Robert Ormandi, Yahoo!

“An ambitiously comprehensive overview of Spark and its diverse ecosystem.”
—Jonathan Miller, Optensity

“An easy-to-follow, step-by-step guide.”
—Gaurav Bhardwaj
3Pillar Global

ISBN-13: 978-1-61729-260-6
ISBN-10: 1-61729-260-5
54999
9 781617 292606

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]