1

Introducing Big Data, Hadoop, and Spark

In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers.

Rear Admiral Grace Murray Hopper, American computer scientist

In This Chapter:

- Introduction to Big Data and the Apache Hadoop project
- Basic overview of the Hadoop core components (HDFS and YARN)
- Introduction to Apache Spark
- Python fundamentals required for PySpark programming, including functional programming basics

The Hadoop and Spark projects are inexorably linked to the Big Data movement. From their early beginnings at search engine providers and in academia to the vast array of current applications that range from data warehousing to complex event processing to machine learning and more, Hadoop and Spark have indelibly altered the data landscape.

This chapter introduces some basic distributed computing concepts, the Hadoop and Spark projects, and functional programming using Python, providing a solid platform to build your knowledge upon as you progress through this book.

Introduction to Big Data, Distributed Computing, and Hadoop

Before discussing Spark, it is important to take a step back and understand the history of what we now refer to as Big Data. To be proficient as a Spark professional, you need to understand not only Hadoop and its use with Spark but also some of the concepts at the core of the Hadoop project, such as data locality, shared nothing, and MapReduce, as these are all applicable and integral to Spark.

A Brief History of Big Data and Hadoop

The set of storage and processing methodologies commonly known as Big Data emerged from the search engine providers in the early 2000s, principally Google and Yahoo!. The search engine providers were the first group of users faced with Internet scale problems, mainly how to process and store indexes of all the documents in the Internet universe. This seemed an insurmountable challenge at the time, even though the entire body of content in the Internet was a fraction of what it is today.

Yahoo! and Google independently set about developing a set of capabilities to meet this challenge. In 2003, Google released a whitepaper titled “The Google File System.” Subsequently, in 2004, Google released another whitepaper, titled “MapReduce: Simplified Data Processing on Large Clusters.” Around the same time, Doug Cutting, who is generally acknowledged as the initial creator of Hadoop, and Mike Cafarella were working on a web crawler project called Nutch, which was based on Cutting’s open source Lucene project (now Apache Lucene). The Google whitepapers inspired Cutting to take the work he had done on the Nutch project and incorporate the storage and processing principles outlined in these whitepapers. The resulting product is what we know today as Hadoop. Later in 2006, Yahoo! decided to adopt Hadoop and hire Doug Cutting to work full time on the project. Hadoop joined the Apache Software Foundation in 2006.

The Apache Software Foundation

> The Apache Software Foundation (ASF) is a nonprofit organization founded in 1999 to provide an open source software structure and framework for developers to contribute to projects. The ASF encourages collaboration and community involvement and protects volunteers from litigation. ASF is premised on the concept of meritocracy, meaning projects are governed by merit.
>
> Contributors are developers who contribute code or documentation to projects. They are typically active on mailing lists and support forums, and they provide suggestions, criticism, and patches to address defects.
>
> Committers are developers whose expertise merits giving them access to a commit code to the main repository for a project. Committers sign a contributor license agreement (CLA) and have an apache.org email address. Committers act as a committee to make decisions about projects.
>
> More information about the Apache Software Foundation can be found at http://apache.org/.

Around the same time the Hadoop project was born, several other technology innovations were afoot, including the following:

- The rapid expansion of ecommerce
- The birth and rapid growth of the mobile Internet
- Blogs and user-driven web content
- Social media

These innovations cumulatively led to an exponential increase in the amount of data generated. This deluge of data accelerated the expansion of the Big Data movement and led to the emergence of other related projects, such as Spark, open source messaging systems such as Kafka, and NoSQL platforms such as HBase and Cassandra, all of which we'll discuss in detail later in this book.

But it all started with Hadoop.

Hadoop Explained

Hadoop is a data storage and processing platform initially based on a central concept: data locality. *Data locality* refers to the pattern of processing data where it resides by bringing the computation to the data rather than the typical pattern of requesting data from its location—for example, a database management system—and sending the data to a remote processing system or host.

With Internet-scale data—Big Data—it is no longer efficient, practical, or even possible in some cases to move the large volumes of data required for processing across the network at compute time.

Hadoop enables large datasets to be processed locally on the nodes of a cluster using a *shared nothing* approach, where each node can independently process a much smaller subset of the entire dataset without needing to communicate with other nodes. This characteristic is enabled through its implementation of a distributed filesystem.

Hadoop is schemaless with respect to its write operations; it is what's known as a *schema-on-read* system. This means it can store and process a wide range of data, from unstructured text documents, to semi-structured JSON (JavaScript Object Notation) or XML documents, to well-structured extracts from relational database systems.

Schema-on-read systems are a fundamental departure from the relational databases we are accustomed to, which are, in contrast, broadly categorized as schema-on-write systems, where data is typically strongly typed and a schema is predefined and enforced upon `INSERT`, `UPDATE`, or `UPSERT` operations.

NoSQL platforms, such as HBase or Cassandra, are also classified as schema-on-read systems. You will learn more about NoSQL platforms in Chapter 6, "SQL and NoSQL Programming with Spark."

Because the schema is not interpreted during write operations to Hadoop, there are no indexes, statistics, or other constructs often employed by database systems to optimize query operations and filter or reduce the amount of data returned to a client. This further necessitates data locality.

Hadoop is designed to find needles in haystacks by dividing and conquering large problems into sets of smaller problems and applying the concepts of data locality and shared nothing. Spark applies the very same concepts.

Core Components of Hadoop

Hadoop has two core components: *HDFS* (Hadoop Distributed File System) and *YARN* (Yet Another Resource Negotiator). HDFS is Hadoop's storage subsystem, whereas YARN can be thought of as Hadoop's processing, or resource scheduling, subsystem (see Figure 1.1).

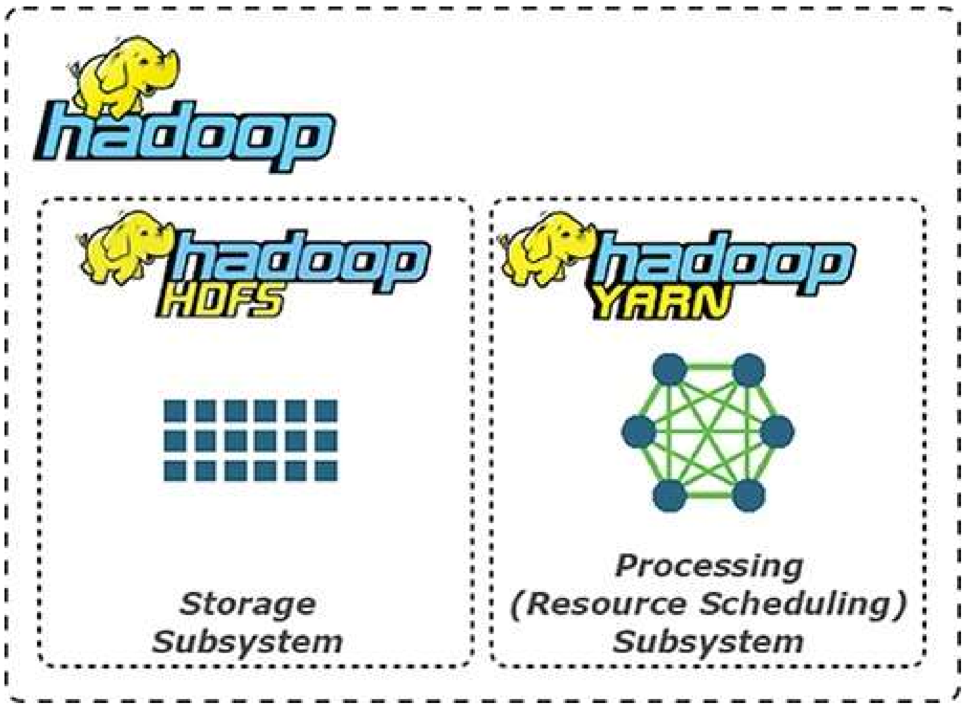


Figure 1.1 Hadoop core components.

Each component is independent of the other and can operate in its own cluster. However, when a HDFS cluster and a YARN cluster are collocated with each other, the combination of both systems is considered to be a Hadoop cluster. Spark can leverage both Hadoop core components, as discussed in more detail later in this chapter.

Cluster Terminology

A *cluster* is a collection of systems that work together to perform functions, such as computational or processing functions. Individual servers within a cluster are referred to as *nodes*.

Clusters can have many topologies and communication models; one such model is the master/slave model. Master/slave is a model of communication whereby one process has control over one or more other processes. In some systems, a master is selected from a group of eligible processes at runtime or during processing, while in other cases—such as with a HDFS or YARN cluster—the master and slave processes are pre-designated static roles for the lifetime of the cluster.

Any other projects that interact or integrate with Hadoop in some way—for instance, data ingestion projects such as Flume or Sqoop or data analysis tools such as Pig or Hive—are called Hadoop “ecosystem” projects. You could consider Spark an ecosystem project, but this is debatable because Spark does not require Hadoop to run.

HDFS: Files, Blocks, and Metadata

HDFS is a virtual filesystem where files are composed of *blocks* distributed across one or more *nodes* of the cluster. Files are split indiscriminately, according to a configured block size upon uploading data into the filesystem, in a process known as *ingestion*. The blocks are then distributed and replicated across cluster nodes to achieve fault tolerance and additional opportunities for processing data locally (the design goal of “bringing the computation to the data”). HDFS blocks are stored and managed on a slave node HDFS cluster process called the *DataNode*.

The DataNode process is the HDFS slave node daemon that runs on one or more nodes of the HDFS cluster. DataNodes are responsible for managing block storage and access for reading and writing of data, as well as for block replication, which is part of the data ingestion process, shown in Figure 1.2.

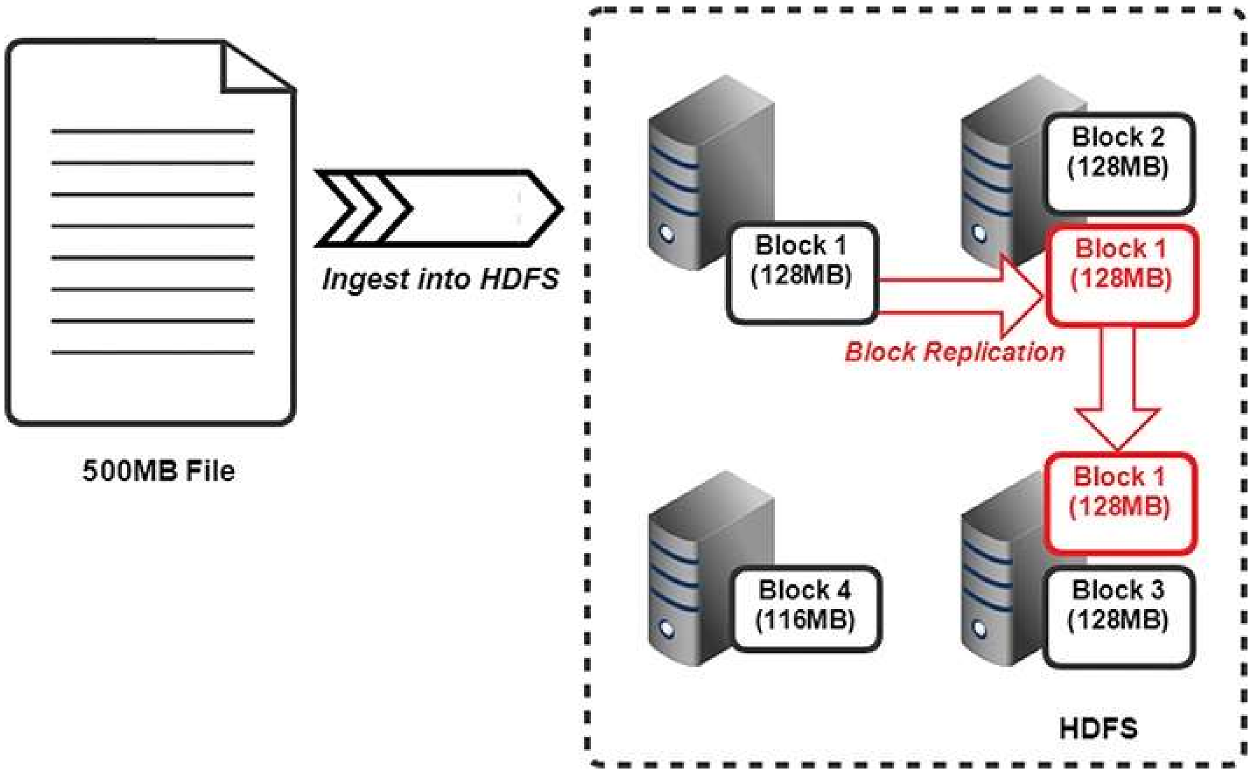


Figure 1.2 HDFS data ingestion, block distribution, and replication.

There are typically many hosts running the DataNode process in a fully distributed Hadoop cluster. Later you will see that the DataNode process provides input data in the form of partitions to distributed Spark worker processes for Spark applications deployed on Hadoop.

The information about the filesystem and its virtual directories, files, and the physical blocks that comprise the files is stored in the filesystem *metadata*. The filesystem metadata is stored in resident memory on the HDFS master node process known as the *NameNode*. The NameNode in a HDFS cluster provides durability to the metadata through a journaling function akin to a relational database transaction log. The NameNode is responsible for providing HDFS clients with block locations for read and write operations, with which the clients communicate directly with the DataNodes for data operations. Figure 1.3 shows the anatomy of an HDFS read operation, and Figure 1.4 shows the anatomy of a write operation in HDFS.

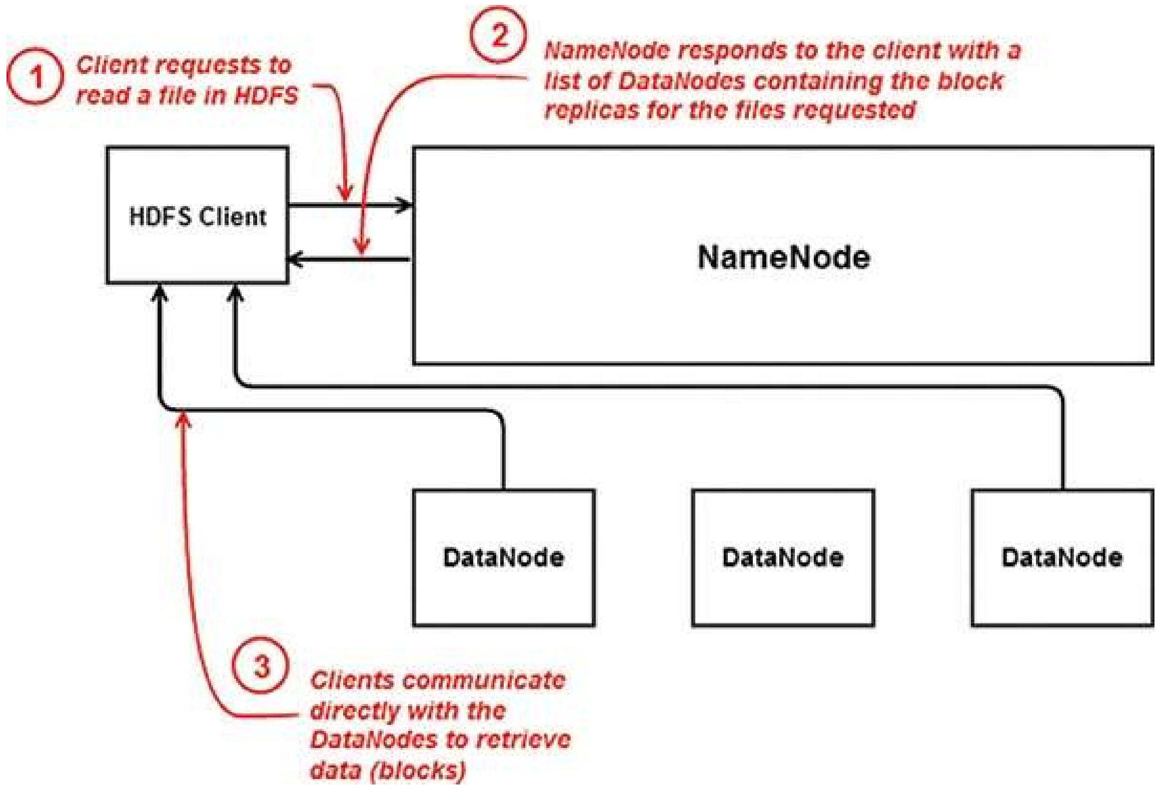

1
Client requests to read a file in HDFS
2
NameNode responds to the client with a list of DataNodes containing the block replicas for the files requested
HDFS Client
NameNode
DataNode
DataNode
DataNode
3
Clients communicate directly with the DataNodes to retrieve data (blocks)

Figure 1.3 Anatomy of an HDFS read operation.

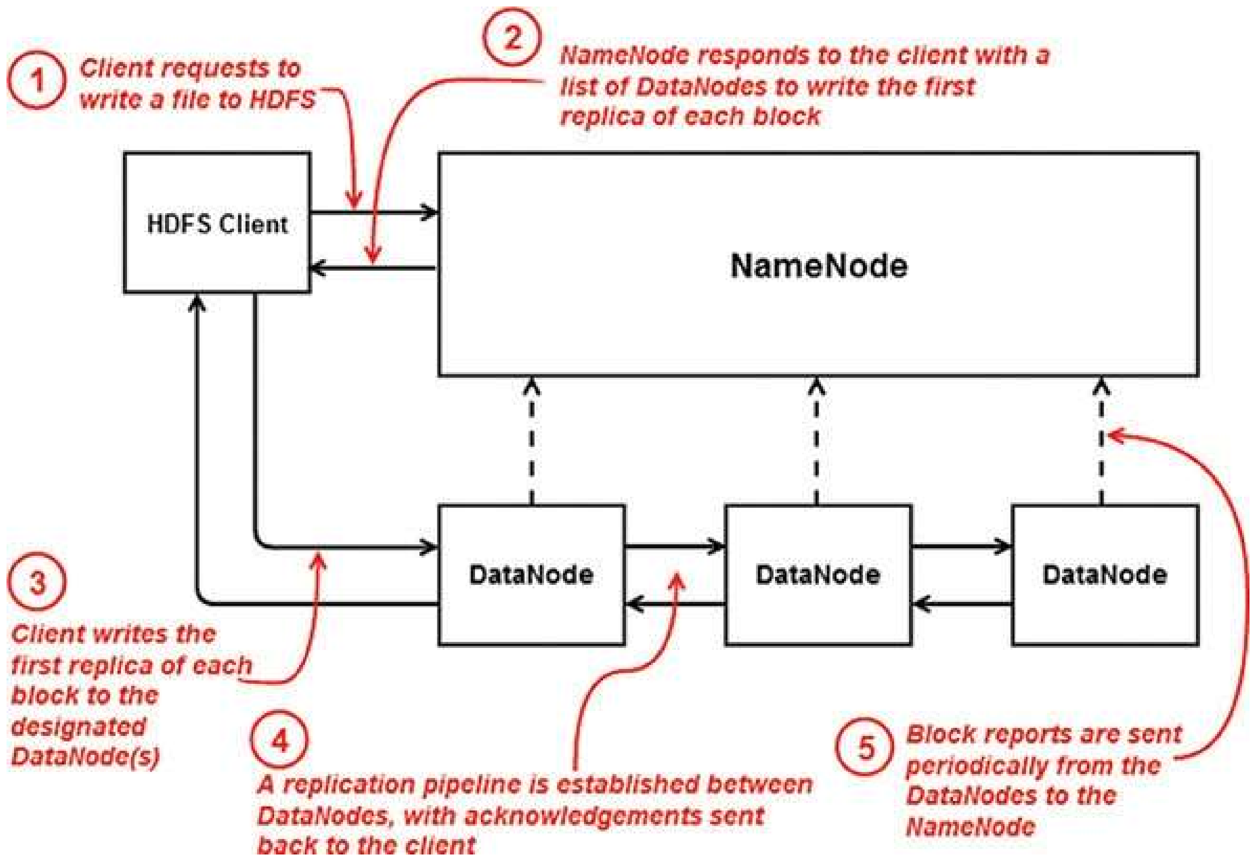

1 Client requests to write a file to HDFS
2 NameNode responds to the client with a list of DataNodes to write the first replica of each block
HDFS Client
NameNode
DataNode
DataNode
DataNode
3 Client writes the first replica of each block to the designated DataNode(s)
4 A replication pipeline is established between DataNodes, with acknowledgements sent back to the client
5 Block reports are sent periodically from the DataNodes to the NameNode

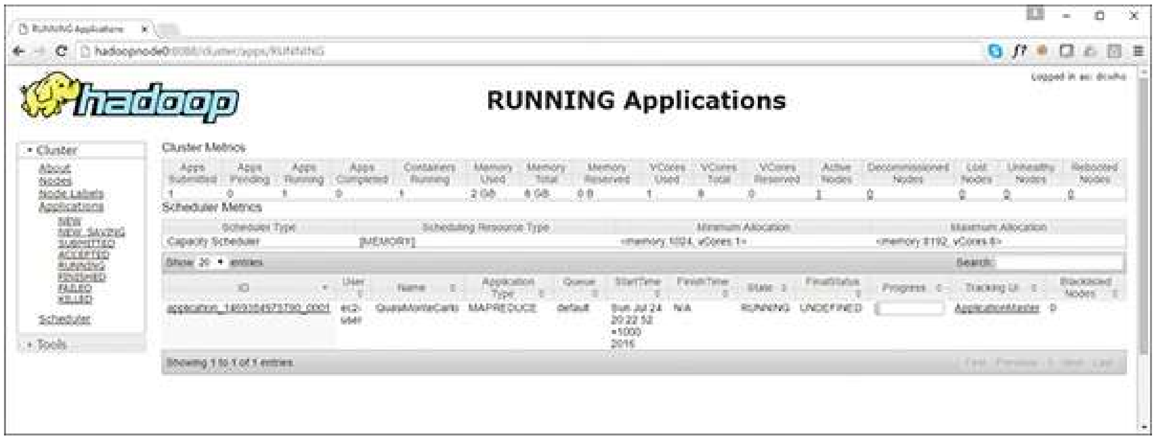
Figure 1.4 Anatomy of an HDFS write operation.

Application Scheduling Using YARN

YARN governs and orchestrates the processing of data in Hadoop, which usually is data sourced from and written to HDFS. The YARN cluster architecture is a master/slave cluster framework like HDFS, with a master node daemon called the *ResourceManager* and one or more slave node daemons called *NodeManagers* running on worker, or slave, nodes in the cluster.

The ResourceManager is responsible for granting cluster compute resources to applications running on the cluster. Resources are granted in units called *containers*, which are predefined combinations of CPU cores and memory. Container allotments, including minimum and maximum thresholds, are configurable on the cluster. Containers are used to isolate resources dedicated to a process or processes.

The ResourceManager also tracks available capacity on the cluster as applications finish and release their reserved resources, and it tracks the status of applications running on the cluster. The ResourceManager serves an embedded web UI on port 8088 of the host running this daemon, which is useful for displaying the status of applications running, completed, or failed on the cluster, as shown in Figure 1.5. You often use this user interface when managing the status of Spark applications running on a YARN cluster.


RUNNING Applications

Logged in as: dr.who

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
1	0	1	0	1	2 GB	8 GB	0 B	1	8	0	1	0	0	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation
Capacity Scheduler	[MEMORY]	<memory:1024, vCores:1>	<memory:8192, vCores:8>

Show 20 entries Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI	Blacklisted Nodes
application_1469354573790_0001	ec2-user	QuasiMonteCarlo	MAPREDUCE	default	Sun Jul 24 20:22:52 +1000 2016	N/A	RUNNING	UNDEFINED		ApplicationMaster	0

Showing 1 to 1 of 1 entries

Figure 1.5 YARN ResourceManager user interface.

Clients submit applications, such as Spark applications, to the ResourceManager; the ResourceManager then allocates the first container on an available NodeManager in the cluster as a delegate process for the application called the *ApplicationMaster*; the ApplicationMaster then negotiates all further containers required to run tasks for the application.

The NodeManager is the slave node YARN daemon that manages containers on the slave node host. Containers are used to execute the tasks involved in an application. As Hadoop's approach to solving large problems is to "divide and conquer," a large problem is deconstructed into a set of tasks, many of which can be run in parallel; recall the concept of shared nothing. These tasks are run in containers on hosts running the NodeManager process.

Most containers simply run tasks. However, the ApplicationMaster has some additional responsibilities for managing an application. As discussed earlier in this chapter, the ApplicationMaster is the first container allocated by the ResourceManager to run on a NodeManager. Its job is to plan the application, including determining what resources are required—often based on how much data is being processed—and to work out resourcing for application stages, which you'll learn about shortly. The ApplicationMaster requests these resources from the ResourceManager on behalf of the application. The ResourceManager grants resources on the same or other NodeManagers to the ApplicationMaster to use for the lifetime of the specific application. The ApplicationMaster—in the case of Spark, as detailed later—monitors the progress of tasks, stages (groups of tasks that can be performed in parallel), and dependencies. The summary information is provided to the ResourceManager to display in its user interface, as shown earlier. A generalization of the YARN application submission, scheduling, and execution process is shown in Figure 1.6.

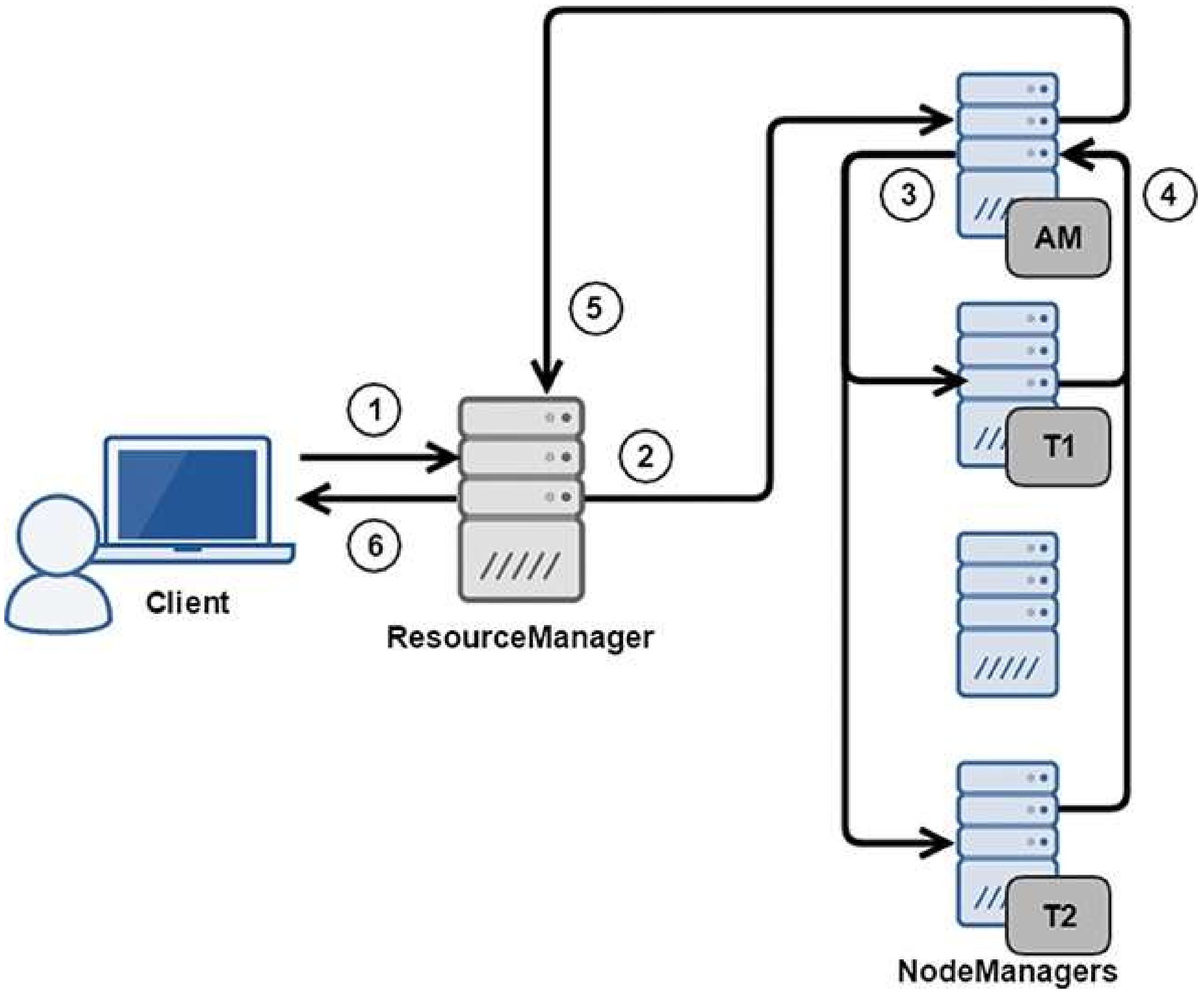


Figure 1.6 YARN application submission, scheduling, and execution (Hadoop 6.6).

The process pictured in Figure 1.6 works as follows:

1. A client submits an application to the ResourceManager.
2. The ResourceManager allocates an ApplicationMaster process on a NodeManager with sufficient capacity to be assigned this role.

3. The ApplicationMaster negotiates task containers with the ResourceManager to be run on NodeManagers—which can include the NodeManager on which the ApplicationMaster is running as well—and dispatches processing to the NodeManagers hosting the task containers for the application.
4. The NodeManagers report their task attempt status and progress to the ApplicationMaster.
5. The ApplicationMaster reports progress and the status of the application to the ResourceManager.
6. The ResourceManager reports application progress, status, and results to the client.

We will explore how YARN is used to schedule and orchestrate Spark programs running on a Hadoop cluster in Chapter 3, "Understanding the Spark Cluster Architecture."

Hadoop MapReduce

Following Google's release of the whitepaper "The Google File System" in 2003, which influenced the HDFS project, Google released another whitepaper, titled "MapReduce: Simplified Data Processing on Large Clusters," in December 2004. The MapReduce whitepaper gives a high-level description of Google's approach to processing—specifically indexing and ranking—large volumes of text data for search engine processing. MapReduce would become the programming model at the core of Hadoop and would ultimately inspire and influence the Spark project.

Introduction to Apache Spark

Apache Spark was created as an alternative to the implementation of MapReduce in Hadoop to gain efficiencies measured in orders of magnitude. Spark also delivers unrivaled extensibility and is effectively a Swiss Army knife for data processing, delivering SQL access, streaming data processing, graph and NoSQL processing, machine learning, and much more.

Apache Spark Background

Apache Spark is an open source distributed data processing project started in 2009 by Matei Zaharia at the University of California, Berkeley, RAD Lab. Spark was created as part of the Mesos research project, designed to look at an alternative resource scheduling and orchestration system to MapReduce. (For more information on Mesos, see http://mesos.apache.org/.)

Using Spark became an alternative to using traditional MapReduce on Hadoop, which was unsuited for interactive queries or real-time, low-latency applications. A major disadvantage of Hadoop's MapReduce implementation was its persistence of intermediate data to disk between the Map and Reduce processing phases.

As an alternative to MapReduce, Spark implements a distributed, fault-tolerant, in-memory structure called a Resilient Distributed Dataset (RDD). Spark maximizes the use of memory across multiple machines, significantly improving overall performance. Spark's reuse of these in-memory structures makes it well suited to iterative machine learning operations as well as interactive queries.

Spark is written in Scala, which is built on top of the Java Virtual Machine (JVM) and Java runtime. This makes Spark a cross-platform application capable of running on Windows as well as Linux; many consider Spark to be the future of data processing in Hadoop.

Spark enables developers to create complex, multi-stage data processing routines, providing a high-level API and fault-tolerant framework that lets programmers focus on logic rather than infrastructure or environmental issues, such as hardware failure.

As a top-level Apache Software Foundation project, Spark has more than 400 individual contributors and committers from companies such as Facebook, Yahoo!, Intel, Netflix, Databricks, and others.

Uses for Spark

Spark supports a wide range of applications, including the following:

- Extract-transform-load (ETL) operations
- Predictive analytics and machine learning
- Data access operations, such as SQL queries and visualizations
- Text mining and text processing

- Real-time event processing
- Graph applications
- Pattern recognition
- Recommendation engines

At the time of this writing, more than 1,500 organizations worldwide are using Spark in production, with some organizations running Spark on hundreds to thousands of cluster nodes against petabytes of data.

Spark's speed and versatility are further complemented by the numerous extensions now included with Spark, including Spark SQL, Spark Streaming, and SparkR, to name a few.

Programming Interfaces to Spark

As mentioned earlier in this chapter, Spark is written in Scala, and it runs in JVMs. Spark provides native support for programming interfaces including the following:

- Scala
- Python (using Python's functional programming operators)
- Java
- SQL
- R

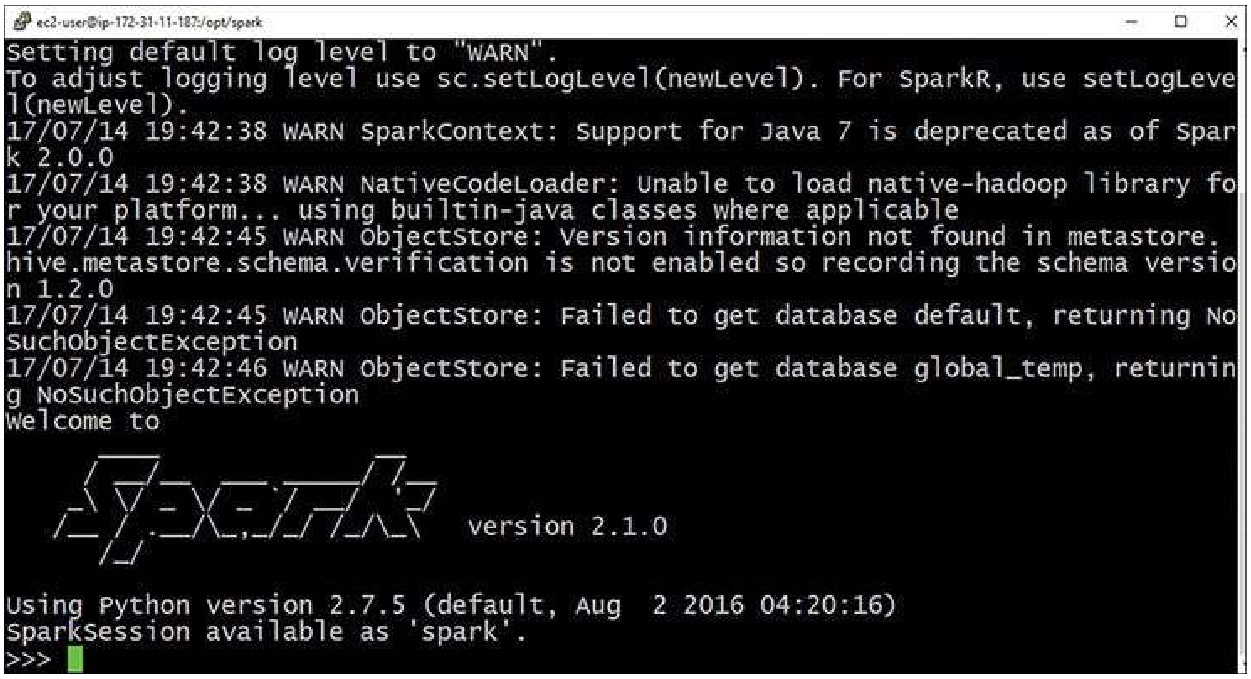
In addition, Spark includes extended support for Clojure and other languages.

Submission Types for Spark Programs

Spark programs can run interactively or as batch jobs, including mini-batch and micro-batch jobs.

Interactive Submission

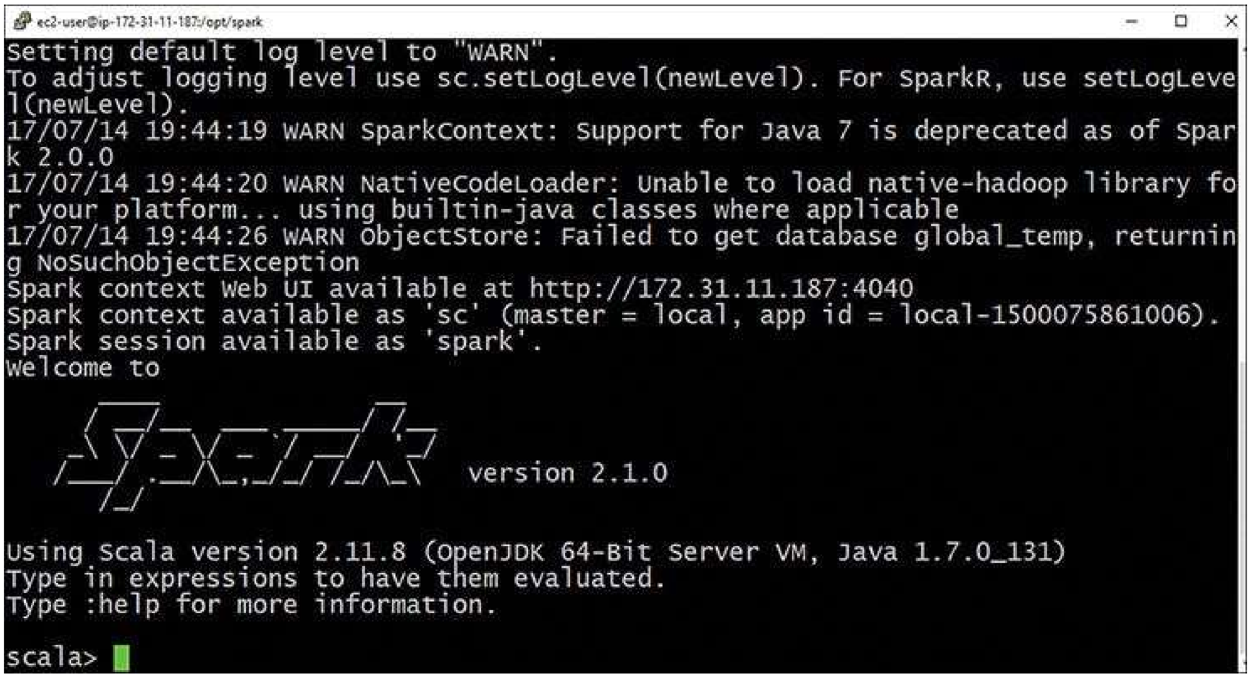
Interactive programming shells are available in Python and Scala. The PySpark and Scala shells are shown in Figures 1.7 and 1.8, respectively.



```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLeve
l(newLevel).
17/07/14 19:42:38 WARN SparkContext: Support for Java 7 is deprecated as of Spar
k 2.0.0
17/07/14 19:42:38 WARN NativeCodeLoader: Unable to load native-hadoop library fo
r your platform... using builtin-java classes where applicable
17/07/14 19:42:45 WARN ObjectStore: Version information not found in metastore.
hive.metastore.schema.verification is not enabled so recording the schema versio
n 1.2.0
17/07/14 19:42:45 WARN ObjectStore: Failed to get database default, returning No
SuchObjectException
17/07/14 19:42:46 WARN ObjectStore: Failed to get database global_temp, returnin
g NoSuchObjectException
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.1.0
      /_/

Using Python version 2.7.5 (default, Aug  2 2016 04:20:16)
SparkSession available as 'spark'.
>>> 
```

Figure 1.7 PySpark shell.



```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLeve
l(newLevel).
17/07/14 19:44:19 WARN SparkContext: Support for Java 7 is deprecated as of Spar
k 2.0.0
17/07/14 19:44:20 WARN NativeCodeLoader: Unable to load native-hadoop library fo
r your platform... using builtin-java classes where applicable
17/07/14 19:44:26 WARN ObjectStore: Failed to get database global_temp, returnin
g NoSuchObjectException
Spark context Web UI available at http://172.31.11.187:4040
Spark context available as 'sc' (master = local, app id = local-1500075861006).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.1.0
      /_/

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.7.0_131)
Type in expressions to have them evaluated.
Type :help for more information.

scala> 
```

Figure 1.8 Scala shell.

Interactive R and SQL shells are included with Spark as well.

Non-interactive or Batch Submission

Non-interactive applications can be submitted using the `spark-submit` command, as shown in Listing 1.1.

Listing 1.1 **Using `spark-submit` to Run a Spark Application Non-interactively**

Click here to view code image

```
$SPARK_HOME/bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master yarn-cluster \
--num-executors 4 \
--driver-memory 10g \
--executor-memory 10g \
--executor-cores 1 \
$SPARK_HOME/examples/jars/spark-examples*.jar 10
```

Input/Output Types for Spark Applications

Although Spark is mostly used to process data in Hadoop, Spark can be used with a multitude of other source and target systems, including the following:

- Local or network filesystems
- Object storage such as Amazon S3 or Ceph
- Relational database systems
- NoSQL stores, including Cassandra, HBase, and others
- Messaging systems such as Kafka

The Spark RDD

We will discuss the Spark Resilient Distributed Dataset (RDD) throughout this book, so it is worthwhile to introduce it now. The Spark RDD, the primary data abstraction structure for Spark applications, is one of the main differentiators between Spark and other cluster computing frameworks. Spark RDDs can be thought of as in-memory collections of data distributed across a cluster. Spark programs using the Spark core API consist of loading input data into an RDD, transforming the RDD into subsequent RDDs, and then storing or presenting the final output for an application from the resulting final RDD. (Don't worry … there is much more about this in upcoming chapters of this book!)

Spark and Hadoop

As noted earlier, Hadoop and Spark are closely related to each other in their shared history and implementation of core parallel processing concepts, such as shared nothing and data locality. Let's look at the ways in which Hadoop and Spark are commonly used together.

HDFS as a Data Source for Spark

Spark can be deployed as a processing framework for data in Hadoop, typically in HDFS. Spark has built-in support for reading and writing to and from HDFS in various file formats, including the following:

- Native text file format
- Sequence file format
- Parquet format

In addition, Spark includes extended support for Avro, ORCFile formats, and others. Reading a file from HDFS using Spark is as easy as this:

Click here to view code image

```
textfile = sc.textFile("hdfs://mycluster/data/file.txt")
```

Writing data from a Spark application to HDFS is as easy as this:

Click here to view code image

```
myRDD.saveAsTextFile("hdfs://mycluster/data/output")
```

YARN as a Resource Scheduler for Spark

YARN is one of the most commonly used process schedulers for Spark applications. Because YARN is usually collocated with HDFS on Hadoop clusters, YARN is a convenient platform for managing Spark applications.

Also, because YARN governs available compute resources across distributed nodes in a Hadoop cluster, it can schedule Spark processing stages to run in parallel wherever possible. Furthermore, where HDFS is used as the input source for a Spark application, YARN can schedule map tasks to take full advantage of data locality, thereby minimizing the amount of data that needs to be transferred across the network during the critical initial stages of processing.

Functional Programming Using Python

Python is an amazingly useful language. Its uses range from automation to web services to machine learning and everything in between. Python has risen to be one of the most widely used languages today.

As a multi-paradigm programming language, Python combines imperative and procedural programming paradigms with full support for the object-oriented and functional paradigms.

The following sections examine the functional programming concepts and elements included in Python, which are integral to Spark's Python API (PySpark)—and are the basis of Spark programming throughout this book—including anonymous functions, common higher-order functions, and immutable and iterable data structures.

Data Structures Used in Functional Python Programming

Python RDDs in Spark are simply representations of distributed collections of Python objects, so it is important to understand the various data structures available in Python.

Lists

Lists in Python are zero-based indexed sequences of mutable values with the first value numbered zero. You can remove or replace elements in a list as well as append elements to the end of a list. Listing 1.2 shows a simple example of a list in Python.

Listing 1.2 **Lists**

Click here to view code image

```
>>> tempc = [38.4, 19.2, 12.8, 9.6]
>>> print(tempc[0])
38.4
>>> print(len(tempc))
4
```

As you can see from Listing 1.2, individual list elements are accessible using the index number in square brackets.

Importantly, lists support the three primary functional programming constructs —`map()`, `reduce()`, and `filter()`—as well as other built-in methods, including `count()`, `sort()`, and more. In this book we will spend a considerable amount of time working with Spark RDDs, which are essentially representations of Python lists. Listing 1.3 provides a basic example of a Python list and a `map()` function. Note that the `map()` function, which we will cover in more detail later, operates on an input list and returns a new list. This example is in pure Python; the equivalent PySpark operation has slightly different syntax.

Listing 1.3 **Python `map()` Function**

Click here to view code image

```
>>> tempf = map(lambda x: (float(9)/5)*x + 32, tempc)
>>> tempf
[101.12, 66.56, 55.040000000000006, 49.28]
```

Although Python lists are mutable by default, list objects contained within Python RDDs in Spark are immutable, as is the case with any objects created within Spark RDDs.

Sets are a similar object type available in Python; they are based upon the set mathematical abstraction. Sets are unordered collections of unique values supporting common mathematical set operations, such as `union()`, `intersection()`, and others.

Tuples

Tuples are an immutable sequence of objects, though the objects contained in a tuple can themselves be immutable or mutable. Tuples can contain different underlying object types, such as a mixture of `string`, `int`, and `float` objects, or they can contain other sequence types, such as sets and other tuples.

For simplicity, think of tuples as being similar to immutable lists. However, they are different constructs and have very different purposes.

Tuples are similar to records in a relational database table, where each record has a structure, and each field defined with an ordinal position in the structure has a meaning. List objects simply have an order, and because they are mutable by default, the order is not directly related to the structure.

Tuples consist of one or more values separated by commas enclosed in parentheses. Elements are accessed from Python tuples similarly to the way they are accessed from lists: using square brackets with a zero-based index referencing the specific element.

Tuple objects have methods for comparing tuples with other tuples, as well as returning the length of a tuple (the number of elements in the tuple). You can also convert a list in Python to a tuple by using the `tuple(list)` function.

Listing 1.4 shows the creation and usage of tuples in native Python.

Listing 1.4 **Tuples**

Click here to view code image

```
>>> rec0 = "Jeff", "Aven", 46
>>> rec1 = "Barack", "Obama", 54
>>> rec2 = "John F", "Kennedy", 46
>>> rec3 = "Jeff", "Aven", 46
>>> rec0
('Jeff', 'Aven', 46)
>>> len(rec0)
3
>>> print("first name: " + rec0[0])
first name: Jeff
# create tuple of tuples
>>> all_recs = rec0, rec1, rec2, rec3
>>> all_recs
(('Jeff', 'Aven', 46), ('Barack', 'Obama', 54),
('John F', 'Kennedy', 46), ('Jeff', 'Aven', 46))
# create list of tuples
>>> list_of_recs = [rec0, rec1, rec2, rec3]
>>> list_of_recs
[('Jeff', 'Aven', 46), ('Barack', 'Obama', 54),
('John F', 'Kennedy', 46), ('Jeff', 'Aven', 46)]
```

As you can see from Listing 1.4, it is very important to distinguish square brackets from parentheses because they have very different structural meanings.

Tuples are integral objects in Spark, as they are typically used to represent key/value pairs, which are often the fundamental unit of data in Spark programming.

Dictionaries

Dictionaries, or *dicts*, in Python are unordered mutable sets of key/value pairs. Dict objects are denoted by curly braces (`{}`), which you can create as empty dictionaries by simply executing a command such as `my_empty_dict = {}`. Unlike with lists and tuples, where an element is accessed by its ordinal position in the sequence (its index), an element in a dict is accessed by its key. A key is separated from its value by a colon (`:`), whereas key/value pairs in a dict are separated by commas.

Dicts are useful because their elements are self-describing rather than relying on a predefined schema or ordinality. Dict elements are accessed by key, as shown in Listing 1.5. This listing also shows how to add or remove elements from a dict, and it shows some useful dict methods, including `keys()`, `values()`, `cmp()`, and `len()`.

Listing 1.5 **Dictionaries**

Click here to view code image

```
>>> dict0 = {'fname':'Jeff', 'lname':'Aven', 'pos':'author'}
>>> dict1 = {'fname':'Barack', 'lname':'Obama', 'pos':'president'}
>>> dict2 = {'fname':'Ronald', 'lname':'Reagan', 'pos':'president'}
>>> dict3 = {'fname':'John', 'mi':'F', 'lname':'Kennedy',
'pos':'president'}
>>> dict4 = {'fname':'Jeff', 'lname':'Aven', 'pos':'author'}
>>> len(dict0)
3
>>> print(dict0['fname'])
```

```
Jeff
>>> dict0.keys()
['lname', 'pos', 'fname']
>>> dict0.values()
['Aven', 'author', 'Jeff']
# compare dictionaries
>>> cmp(dict0, dict1)
1 ## keys match but values dont
>>> cmp(dict0, dict4)
0 ## all key value pairs match
>>> cmp(dict1, dict2)
-1 ## some key value pairs match
```

Dicts can be used as immutable objects within a Python RDD.

Python Object Serialization

Serialization is the process of converting an object into a structure that can be unpacked (deserialized) at a later point in time on the same system or on a different system.

Serialization, or the ability to serialize and deserialize data, is a necessary function of any distributed processing system and features heavily throughout the Hadoop and Spark projects.

JSON

JSON (JavaScript Object Notation) is a common serialization format. JSON has extended well beyond JavaScript and is used in a multitude of platforms, with support in nearly every programming language. It is a common response structure returned from web services.

JSON is supported natively in Python using the `json` package. A package is a set of libraries or a collection of modules (which are essentially Python files). The `json` package is used to encode and decode JSON. A JSON object consists of key/value pairs (dictionaries) and/or arrays (lists), which can be nested within each other. The Python JSON object includes methods for searching, adding, and deleting keys; updating values; and printing objects. Listing 1.6 demonstrates creating a JSON object in Python and performing various actions.

Listing 1.6 **Using a JSON Object in Python**

Click here to view code image

```
>>> import json
>>> from pprint import pprint
>>> json_str = '''{
... "people" : [
... {"fname": "Jeff",
... "lname": "Aven",
... "tags": ["big data","hadoop"]},
... {"fname": "Doug",
... "lname": "Cutting",
... "tags": ["hadoop","avro","apache","java"]},
... {"fname": "Martin",
... "lname": "Odersky",
... "tags": ["scala","typesafe","java"]},
... {"fname": "John",
... "lname": "Doe",
... "tags": []}
... ]}'''
>>> people = json.loads(json_str)
>>> len(people["people"])
4
>>> print(people["people"][0]["fname"])
Jeff
# add tag item to the first person
people["people"][0]["tags"].append(u'spark')
# delete the fourth person
del people["people"][3]
# "pretty print" json object
pprint(people)
{u'people': [{u'fname': u'Jeff',
              u'lname': u'Aven',
              u'tags': [u'big data', u'hadoop', u'spark']},
             {u'fname': u'Doug',
              u'lname': u'Cutting',
              u'tags': [u'hadoop', u'avro', u'apache', u'java']},
             {u'fname': u'Martin',
              u'lname': u'Odersky',
              u'tags': [u'scala', u'typesafe', u'java']}]}
```

JSON objects can be used within RDDs in PySpark; we will look at this in detail a bit later in this book.

Pickle

Pickle is a serialization method that is proprietary to Python. Pickle is faster than JSON. However, it lacks the portability of JSON, which is a universally interchangeable serialization format.

The Python `pickle` module converts a Python object or objects into a byte stream that can be transmitted, stored, and reconstructed into its original state.

`cPickle`, as the name suggests, is implemented in C instead of Python, and thus it is much faster than the Python implementation. There are some limitations, however. The `cPickle` module does not support subclassing, which is possible using the `pickle` module. Pickling and unpickling an object in Python is a straightforward process, as shown in Listing 1.7. Notice that the load and dump idioms are analogous to the way you serialize and deserialize objects using JSON. The `pickle.dump` approach saves the pickled object to a file, whereas `pickle.dumps` returns the pickled representation of the object as a string that may look strange, although it is not designed to be human readable.

Listing 1.7 **Object Serialization Using Pickle in Python**

Click here to view code image

```
>>> import cPickle as pickle
>>> obj = { "fname": "Jeff", \
... "lname": "Aven", \
... "tags": ["big data","hadoop"]}
>>> str_obj = pickle.dumps(obj)
>>> pickled_obj = pickle.loads(str_obj)
>>> print(pickled_obj["fname"])
Jeff
>>> pickled_obj["tags"].append('spark')
>>> print(str(pickled_obj["tags"]))
['big data', 'hadoop', 'spark']
# dump pickled object to a string
>>> pickled_obj_str = pickle.dumps(pickled_obj)
# dump pickled object to a pickle file
```

```
>>> pickle.dump(pickled_obj, open('object.pkl', 'wb'))
```

The PickleSerializer is used in PySpark to load objects into a pickled format and to unpickle objects; this includes reading preserialized objects from other systems, such as SequenceFiles in Hadoop, and converting them into a format that is usable by Python.

PySpark includes two methods for handling pickled input and output files: `pickleFile` and `saveAsPickleFile`. `pickleFile` is an efficient format for storing and transferring files between PySpark processes. We will examine these methods later in this book.

Aside from its explicit use by developers, pickling is also used by many internal Spark processes in the execution of Spark applications in Python.

Python Functional Programming Basics

Python’s functional support embodies all of the functional programming paradigm characteristics that you would expect, including the following:

- Functions as first-class objects and the fundamental unit of programming
- Functions with input and output only (Statements, which could result in side effects, are not allowed.)
- Support for higher-order functions
- Support for anonymous functions

The next few sections look at some of functional programming concepts and their implementation in Python.

Anonymous Functions and the `lambda` Syntax

Anonymous functions, or unnamed functions, are a consistent feature of functional programming languages such as Lisp, Scala, JavaScript, Erlang, Clojure, Go, and many more.

Anonymous functions in Python are implemented using the `lambda` construct rather than using the `def` keyword for named functions. Anonymous functions accept any number of input arguments but return just one value. This value could be another function, a scalar value, or a data structure such as a list.

Listing 1.8 shows two similar functions; one is a named function and one is an anonymous function.

Listing 1.8 **Named Functions and Anonymous Functions in Python**

Click here to view code image

```
# named function
>>> def plusone(x): return x+1
...
>>> plusone(1)
2
>>> type(plusone)
<type 'function'>
# anonymous function
>>> plusonefn = lambda x: x+1
>>> plusonefn(1)
2
>>> type(plusonefn)
<type 'function'>
>>> plusone.func_name
'plusone'
>>> plusonefn.func_name
'<lambda>'
```

As you can see in Listing 1.8, the named function `plusone` keeps a reference to the function name, whereas the anonymous function `plusonefn` keeps a `<lambda>` name reference.

Named functions can contain statements such as `print`, but anonymous functions can contain only a single or compound expression, which could be a call to another named function that is in scope. Named functions can also use the `return` statement, which is not supported with anonymous functions.

The true power of anonymous functions is evident when you look at higher-order functions, such as `map()`, `reduce()`, and `filter()`, and start chaining single-use functions together in a processing pipeline, as you do in Spark.

Higher-Order Functions

A higher-order function accepts functions as arguments and can return a function as a result. `map()`, `reduce()`, and `filter()` are examples of higher-order functions. These functions accept a function as an argument.

The `flatMap()`, `filter()`, `map()`, and `reduceByKey()` functions in Listing 1.9 are all examples of higher-order functions because they accept and expect an anonymous function as input.

Listing 1.9 **Examples of Higher-Order Functions in Spark**

Click here to view code image

```
>>> lines = sc.textFile("file:///opt/spark/licenses")
>>> counts = lines.flatMap(lambda x: x.split(' ')) \
... .filter(lambda x: len(x) > 0) \
... .map(lambda x: (x, 1)) \
... .reduceByKey(lambda x, y: x + y) \
... .collect()
>>> for (word, count) in counts:
...     print("%s: %i" % (word, count))
```

Functions that return functions as a return value are also considered higher-order functions. This characteristic defines callbacks implemented in asynchronous programming.

Don't stress … We will cover all these functions in detail in Chapter 4, "Learning Spark Programming Basics." For now it is only important to understand the concept of higher-order functions.

Closures

Closures are function objects that enclose the scope at the time they were instantiated. This can include any external variables or functions used when the function was created. Closures "remember" the values by enclosing the scope.

Listing 1.10 is a simple example of closures in Python.

Listing 1.10 **Closures in Python**

Click here to view code image

```
>>> def generate_message(concept):
...     def ret_message():
...             return 'This is an example of ' + concept
...     return ret_message
...
>>> call_func = generate_message('closures in Python')
>>> call_func
<function ret_message at 0x7fd138aa55f0>
>>> call_func()
'This is an example of closures in Python'
# inspect closure
>>> call_func.__closure__
(<cell at 0x7fd138aaa638: str object at 0x7fd138aaa688>,)
>>> type(call_func.__closure__[0])
<type 'cell'>
>>> call_func.__closure__[0].cell_contents
'closures in Python'
# delete function
del generate_message
# call closure again
call_func()
'This is an example of closures in Python'
# the closure still works!
```

In Listing 1.10, the function `ret_message()` is the closure, and the value for `concept` is enclosed in the function scope. You can use the `__closure__` function member to see information about the closure. The references enclosed in the function are stored in a tuple of cells. You can access the cell contents by using the `cell_contents` function, as shown in this listing. To prove the concept of closures, you can delete the outer function, `generate_message`, and find that the referencing function, `call_func`, still works.

The concept of closures is important to grasp because closures can be of significant benefit in a distributed Spark application. Conversely, closures can have a detrimental impact as well, depending on how the function you are using is constructed and called.

Summary

In this chapter you have gained an understanding of the history, motivation, and uses of Spark, as well as a solid background on Hadoop, a project that is directly correlated to Spark. You have learned the basic fundamentals or HDFS and YARN, the core components of Hadoop, and how these components are used by Spark. This chapter discussed the beginnings of the Spark project along with how Spark is used. This chapter also provided a primer on basic functional programming concepts and their implementations in Python and PySpark. Many of the concepts introduced in this chapter are referenced throughout the remainder of this book.