Chapter 2: Downloading Apache Spark and Getting Started

In this chapter, we will get you set up with Spark and walk through three simple steps you can take to get started writing your first standalone application.

We will use local mode, where all the processing is done on a single machine in a Spark shell—this is an easy way to learn the framework, providing a quick feedback loop for iteratively performing Spark operations. Using a Spark shell, you can prototype Spark operations with small data sets before writing a complex Spark application, but for large data sets or real work where you want to reap the benefits of distributed execution, local mode is not suitable—you'll want to use the YARN or Kubernetes deployment modes instead.

While the Spark shell only supports Scala, Python, and R, you can write a Spark application in any of the supported languages (including Java) and issue queries in Spark SQL. We do expect you to have some familiarity with the language of your choice.

Step 1: Downloading Apache Spark

To get started, go to the Spark download page, select "Pre-built for Apache Hadoop 2.7" from the drop-down menu in step 2, and click the "Download Spark" link in step 3 (Figure 2-1).

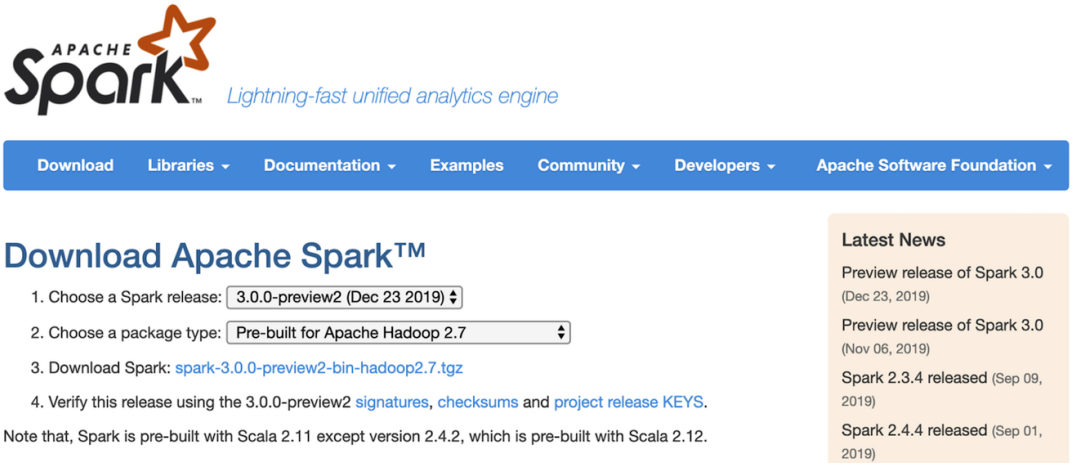

Figure 2-1. The Apache Spark download page

This will download the tarball *spark-3.0.0-preview2-bin-hadoop2.7.tgz*, which contains all the Hadoop-related binaries you will need to run Spark in local mode on your laptop. Alternatively, if you're going to install it on an existing HDFS or Hadoop installation, you can select the matching Hadoop version from the drop-down menu. How to build from source is beyond the scope of this book, but you can read more about it in the documentation.

NOTE

At the time this book went to press Apache Spark 3.0 was still in preview mode, but you can download the latest Spark 3.0 using the same download method and instructions.

Since the release of Apache Spark 2.2, developers who only care about learning Spark in Python have the option of installing PySpark from the PyPI repository. If you only program in Python, you don't have to install all the other libraries necessary to run Scala, Java, or R; this makes the binary smaller. To install PySpark from PyPI, just run `pip install pyspark` .

There are some extra dependencies that can be installed for SQL, ML, and MLlib, via `pip install pyspark[sql,ml,mllib]` (or `pip install pyspark[sql]` if you only want the SQL dependencies).

NOTE

You will need to install Java 8 or above on your machine and set the `JAVA_HOME` environment variable. See the documentation for instructions on how to download and install Java.

If you want to run R in an interpretive shell mode, you must install R and then run `sparkR`. To do distributed computing with R, you can also use the open source project `sparklyr`, created by the R community.

Spark's Directories and Files

We assume that you are running a version of the Linux or macOS operating system on your laptop or cluster, and all the commands and instructions in this book will be in that flavor. Once you have finished downloading the tarball, `cd` to the downloaded directory, extract the tarball contents with `tar -xf spark-3.0.0-preview2-bin-hadoop2.7.tgz`, and `cd` into that directory and take a look at the contents:

```
$ cd spark-3.0.0-preview2-bin-hadoop2.7
$ ls
LICENSE   R          RELEASE   conf    examples   kubernetes  python   yarn
NOTICE    README.md  bin       data    jars       licenses    sbin
```

Let's briefly summarize the intent and purpose of some of these files and directories. New items were added in Spark 2.x and 3.0, and the contents of some of the existing files and directories were changed too:

README.md

: This file contains new detailed instructions on how to use Spark shells, build Spark from source, run standalone Spark examples, peruse links to Spark documentation and configuration guides, and contribute to Spark.

bin

: This directory, as the name suggests, contains most of the scripts you'll employ to interact with Spark, `including` the `Spark shells` (`spark-sql`, `pyspark`, `spark-shell`, and `sparkR`). We will use these shells and executables in this directory later in this chapter to submit a standalone Spark application using `spark-submit`, and write a script that builds and pushes Docker images when running Spark with Kubernetes support.

sbin

Most of the scripts in this directory are administrative in purpose, for starting and stopping Spark components in the cluster in its various deployment modes. For details on the deployment modes, see the cheat sheet in Table 1-1 in Chapter 1.

kubernetes

Since the release of Spark 2.4, this directory contains Dockerfiles for creating Docker images for your Spark distribution on a Kubernetes cluster. It also contains a file providing instructions on how to build the Spark distribution before building your Docker images.

data

This directory is populated with **.txt* files that serve as input for Spark's components: MLlib, Structured Streaming, and GraphX.

examples

For any developer, two imperatives that ease the journey to learning any new platform are loads of "how-to" code examples and comprehensive documentation. Spark provides examples for Java, Python, R, and Scala, and you'll want to employ them when learning the framework. We will allude to some of these examples in this and subsequent chapters.

Step 2: Using the Scala or PySpark Shell

As mentioned earlier, Spark comes with four widely used interpreters that act like interactive "shells" and enable ad hoc data analysis: `pyspark`, `spark-shell`, `spark-sql`, and `sparkR`. In many ways, their interactivity imitates shells you'll already be familiar with if you have experience with Python, Scala, R, SQL, or Unix operating system shells such as bash or the Bourne shell.

These shells have been augmented to support connecting to the cluster and to allow you to load distributed data into Spark workers' memory. Whether you are dealing with gigabytes of data or small data sets, Spark shells are conducive to learning Spark quickly.

To start PySpark, `cd` to the *bin* directory and launch a shell by typing **`pyspark`**. If you have installed PySpark from PyPI, then just typing **`pyspark`** will suffice:

```
$ pyspark
Python 3.7.3 (default, Mar 27 2019, 09:23:15)
[Clang 10.0.1 (clang-1001.0.46.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
20/02/16 19:28:48 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform... using builtin-java classes where applicable
Welcome to

      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.0.0-preview2
      /_/

Using Python version 3.7.3 (default, Mar 27 2019 09:23:15)
SparkSession available as 'spark'.
>>> spark.version
'3.0.0-preview2'
>>>
```

To start a similar Spark shell with Scala, `cd` to the *bin* directory and type **`spark-shell`**:

```
$ spark-shell
20/05/07 19:30:26 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform... using builtin-java classes where applicable
Spark context Web UI available at http://10.0.1.7:4040
Spark context available as 'sc' (master = local[*], app id = local-158191023190
Spark session available as 'spark'.
Welcome to


      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.0.0-preview2
      /_/

Using Scala version 2.12.10 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_241)
Type in expressions to have them evaluated.
Type :help for more information.
scala> spark.version
res0: String = 3.0.0-preview2
scala>
```

Using the Local Machine

Now that you've downloaded and installed Spark on your local machine, for the remainder of this chapter you'll be using Spark interpretive shells locally. That is, Spark will be running in local mode.

NOTE

Refer to Table 1-1 in Chapter 1 for a reminder of which components run where in local mode.

As noted in the previous chapter, Spark computations are expressed as operations. These operations are then converted into low-level RDD-based bytecode as tasks, which are distributed to Spark's executors for execution.

Let's look at a short example where we read in a text file as a DataFrame, show a sample of the strings read, and count the total number of lines in the file. This simple example illustrates the use of the high-level Structured APIs, which we will cover in the next chapter. The `show(10, false)` operation on the DataFrame only displays the first 10 lines without truncating; by default the `truncate` Boolean flag is `true`. Here's what this looks like in the Scala shell:

```
scala> val strings = spark.read.text("../README.md")
strings: org.apache.spark.sql.DataFrame = [value: string]

scala> strings.show(10, false)
+------------------------------------------------------------------------------+
|value                                                                         |
+------------------------------------------------------------------------------+
|# Apache Spark                                                                |
|                                                                              |
|Spark is a unified analytics engine for large-scale data processing. It       |
|provides high-level APIs in Scala, Java, Python, and R, and an optimized      |
|engine that supports general computation graphs for data analysis. It also    |
|supports a rich set of higher-level tools including Spark SQL for SQL and     |
|DataFrames, MLlib for machine learning, GraphX for graph processing,          |
| and Structured Streaming for stream processing.                              |
|                                                                              |
```

```
|<https://spark.apache.org/>
+------------------------------------------------------------------------+
only showing top 10 rows

scala> strings.count()
res2: Long = 109
scala>
```

Quite simple. Let's look at a similar example using the Python interpretive shell, `pyspark`:

```
$ pyspark
Python 3.7.3 (default, Mar 27 2019, 09:23:15)
[Clang 10.0.1 (clang-1001.0.46.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal
reflective access operations
WARNING: All illegal access operations will be denied in a future release
20/01/10 11:28:29 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
Welcome to

      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.0.0-preview2
      /_/

Using Python version 3.7.3 (default, Mar 27 2019 09:23:15)
SparkSession available as 'spark'.
>>> strings = spark.read.text("../README.md")
>>> strings.show(10, truncate=False)
+------------------------------------------------------------------------+
|value
+------------------------------------------------------------------------+
|# Apache Spark
|
```

```
|Spark is a unified analytics engine for large-scale data processing. It
|provides high-level APIs in Scala, Java, Python, and R, and an optimized
|engine that supports general computation graphs for data analysis. It also
|supports a rich set of higher-level tools including Spark SQL for SQL and
|DataFrames, MLlib for machine learning, GraphX for graph processing,
|and Structured Streaming for stream processing.
|
|<https://spark.apache.org/>
+--------------------------------------------------------------------------------
only showing top 10 rows

>>> strings.count()
109
>>>
```

To exit any of the Spark shells, press Ctrl-D. As you can see, this rapid interactivity with Spark shells is conducive not only to rapid learning but to rapid prototyping, too.

In the preceding examples, notice the API syntax and signature parity across both Scala and Python. Throughout Spark's evolution from 1.x, that has been one (among many) of the enduring improvements.

Also note that we used the high-level Structured APIs to read a text file into a Spark DataFrame rather than an RDD. Throughout the book, we will focus more on these Structured APIs; since Spark 2.x, RDDs are now consigned to low-level APIs.

NOTE

Every computation expressed in high-level Structured APIs is decomposed into low-level optimized and generated RDD operations and then converted into Scala bytecode for the executors' JVMs. This generated RDD operation code is not accessible to users, nor is it the same as the user-facing RDD APIs.

Step 3: Understanding Spark Application Concepts

Now that you have downloaded Spark, installed it on your laptop in standalone mode, launched a Spark shell, and executed some short code examples interactively, you're ready to take the final step.

To understand what's happening under the hood with our sample code, you'll need to be familiar with some of the key concepts of a Spark application and how the code is transformed and executed as tasks across the Spark executors. We'll begin by defining some important terms:

Application

A user program built on Spark using its APIs. It consists of a driver program and executors on the cluster.

`SparkSession`

An object that provides a point of entry to interact with underlying Spark functionality and allows programming Spark with its APIs. In an interactive Spark shell, the Spark driver instantiates a `SparkSession` for you, while in a Spark application, you create a `SparkSession` object yourself.

Job

A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g., `save()`, `collect()`).

Stage

Each job gets divided into smaller sets of tasks called stages that depend on each other.

Task

A single unit of work or execution that will be sent to a Spark executor.

Let's dig into these concepts in a little more detail.

Spark Application and SparkSession

At the core of every Spark application is the Spark driver program, which creates a `SparkSession` object. When you're working with a Spark shell, the driver is part of the shell and the `SparkSession` object (accessible via

the variable `spark`) is created for you, as you saw in the earlier examples when you launched the shells.

In those examples, because you launched the Spark shell locally on your laptop, all the operations ran locally, in a single JVM. But you can just as easily launch a Spark shell to analyze data in parallel on a cluster as in local mode. The commands `spark-shell --help` or `pyspark --help` will show you how to connect to the Spark cluster manager. Figure 2-2 shows how Spark executes on a cluster once you've done this.

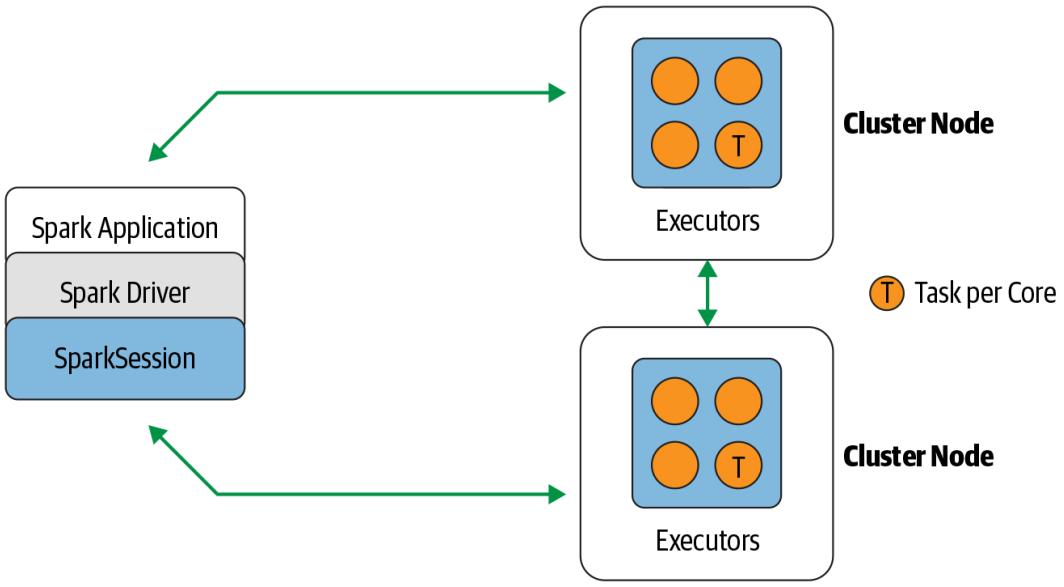

Figure 2-2. Spark components communicate through the Spark driver in Spark's distributed architecture

Once you have a `SparkSession`, you can program Spark using the APIs to perform Spark operations.

Spark Jobs

During interactive sessions with Spark shells, the driver converts your Spark application into one or more Spark jobs (Figure 2-3). It then transforms each job into a DAG. This, in essence, is Spark's execution plan, where each node within a DAG could be a single or multiple Spark stages.

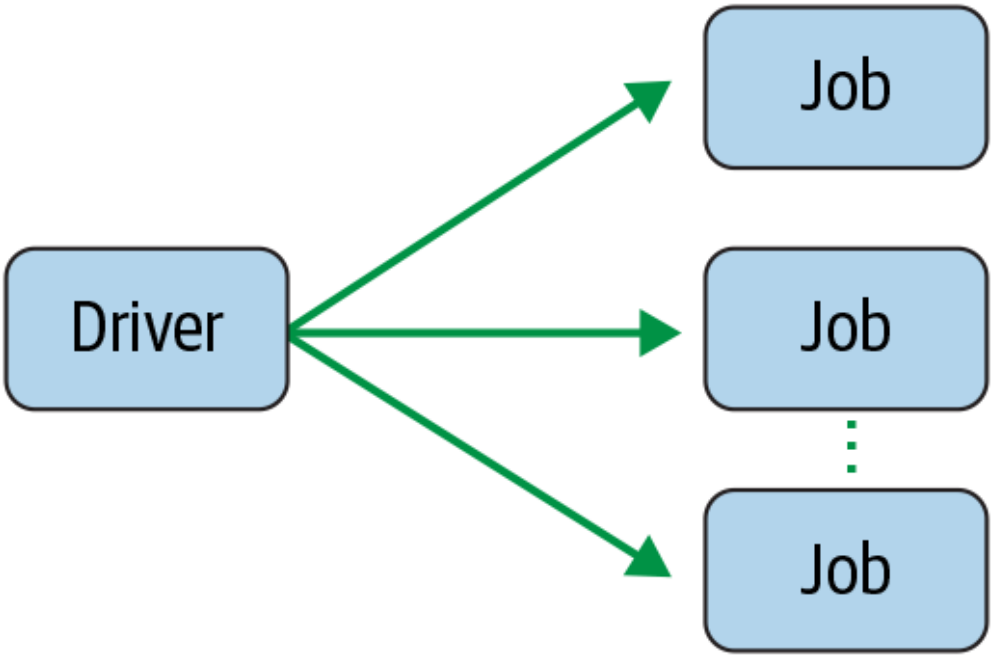

Figure 2-3. Spark driver creating one or more Spark jobs

Spark Stages

As part of the DAG nodes, stages are created based on what operations can be performed serially or in parallel (Figure 2-4). Not all Spark operations can happen in a single stage, so they may be divided into multiple stages. Often stages are delineated on the operator's computation boundaries, where they dictate data transfer among Spark executors.

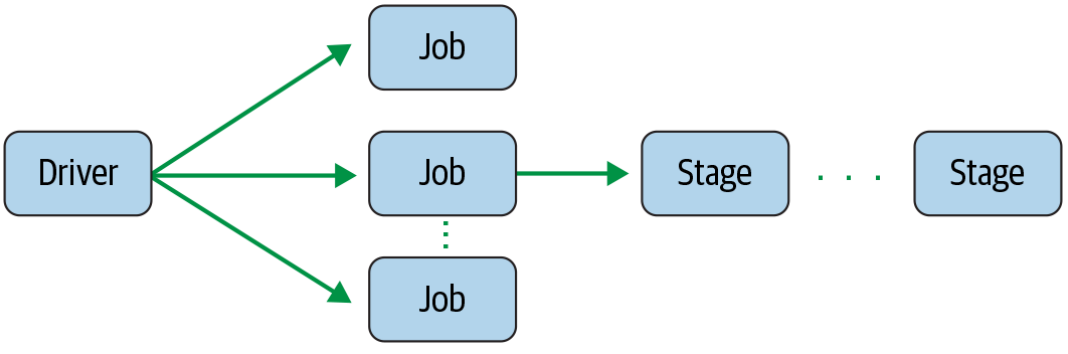

Figure 2-4. Spark job creating one or more stages

Spark Tasks

Each stage is comprised of Spark tasks (a unit of execution), which are then federated across each Spark executor; each task maps to a single core and works on a single partition of data (Figure 2-5). As such, an ex-

ecutor with 16 cores can have 16 or more tasks working on 16 or more partitions in parallel, making the execution of Spark's tasks exceedingly parallel!

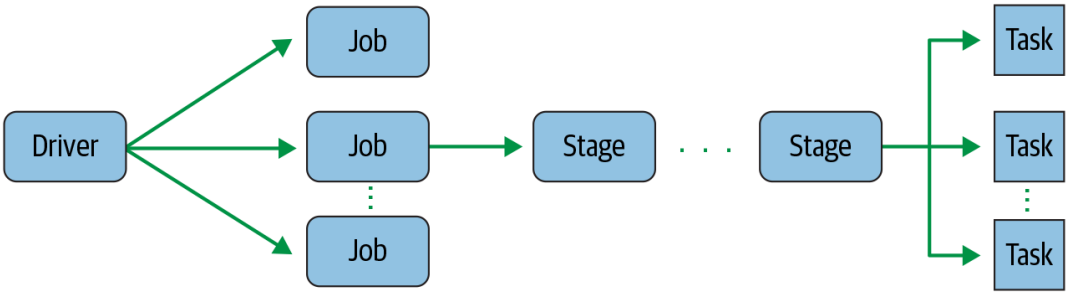

Figure 2-5. Spark stage creating one or more tasks to be distributed to executors

Transformations, Actions, and Lazy Evaluation

Spark operations on distributed data can be classified into two types: *transformations* and *actions*. Transformations, as the name suggests, transform a Spark DataFrame into a new DataFrame without altering the original data, giving it the property of immutability. Put another way, an operation such as `select()` or `filter()` will not change the original DataFrame; instead, it will return the transformed results of the operation as a new DataFrame.

All transformations are evaluated lazily. That is, their results are not computed immediately, but they are recorded or remembered as a *lineage*. A recorded lineage allows Spark, at a later time in its execution plan, to rearrange certain transformations, coalesce them, or optimize transformations into stages for more efficient execution. Lazy evaluation is Spark's strategy for delaying execution until an action is invoked or data is "touched" (read from or written to disk).

An action triggers the lazy evaluation of all the recorded transformations. In Figure 2-6, all transformations T are recorded until the action A is invoked. Each transformation T produces a new DataFrame.

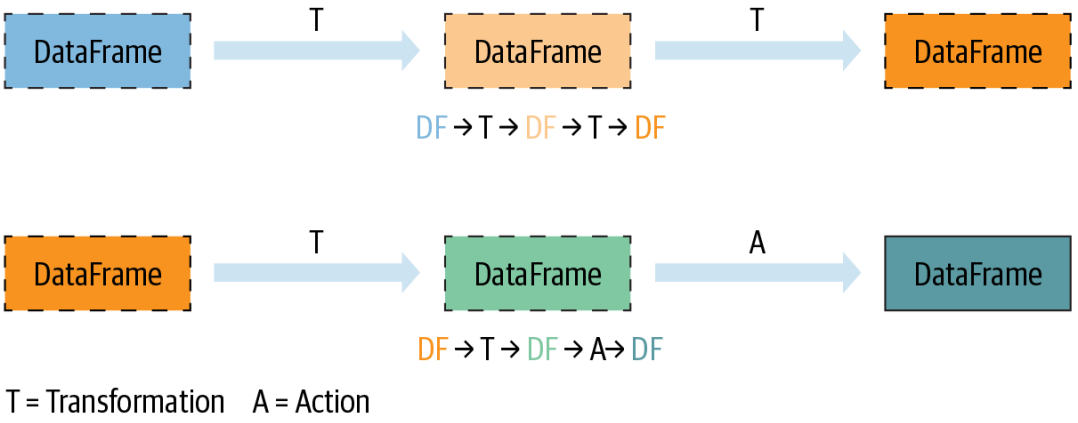

Figure 2-6. Lazy transformations and eager actions

While lazy evaluation allows Spark to optimize your queries by peeking into your chained transformations, lineage and data immutability provide fault tolerance. Because Spark records each transformation in its lineage and the DataFrames are immutable between transformations, it can reproduce its original state by simply replaying the recorded lineage, giving it resiliency in the event of failures.

Table 2-1 lists some examples of transformations and actions.

Table 2-1. Transformations and actions as Spark operations

Transformations	Actions
`orderBy()`	`show()`
`groupBy()`	`take()`
`filter()`	`count()`
`select()`	`collect()`
`join()`	`save()`

The actions and transformations contribute to a Spark query plan, which we will cover in the next chapter. Nothing in a query plan is executed until an action is invoked. The following example, shown both in Python and Scala, has two transformations— `read()` and `filter()` —and one

action— `count()` . The action is what triggers the execution of all transformations recorded as part of the query execution plan. In this example, nothing happens until `filtered.count()` is executed in the shell:

```
# In Python
>>> strings = spark.read.text("../README.md")
>>> filtered = strings.filter(strings.value.contains("Spark"))
>>> filtered.count()
20
```

```
// In Scala
scala> import org.apache.spark.sql.functions._
scala> val strings = spark.read.text("../README.md")
scala> val filtered = strings.filter(col("value").contains("Spark"))
scala> filtered.count()
res5: Long = 20
```

Narrow and Wide Transformations

As noted, transformations are operations that Spark evaluates lazily. A huge advantage of the lazy evaluation scheme is that Spark can inspect your computational query and ascertain how it can optimize it. This optimization can be done by either joining or pipelining some operations and assigning them to a stage, or breaking them into stages by determining which operations require a shuffle or exchange of data across clusters.

Transformations can be classified as having either *narrow dependencies* or *wide dependencies*. Any transformation where a single output partition can be computed from a single input partition is a *narrow* transformation. For example, in the previous code snippet, `filter()` and `contains()` represent narrow transformations because they can operate on a single partition and produce the resulting output partition without any exchange of data.

However, transformations such as `groupBy()` or `orderBy()` instruct Spark to perform wide transformations, where data from other partitions is read in, combined, and written to disk. If we were to sort the `filtered` DataFrame from the preceding example by calling `.orderBy()` , each partition will be locally sorted, but we need to force a shuffle of data from

each of the executor's partitions across the cluster to sort all of the records. In contrast to narrow transformations, wide transformations require output from other partitions to compute the final aggregation.

Figure 2-7 illustrates the two types of dependencies.

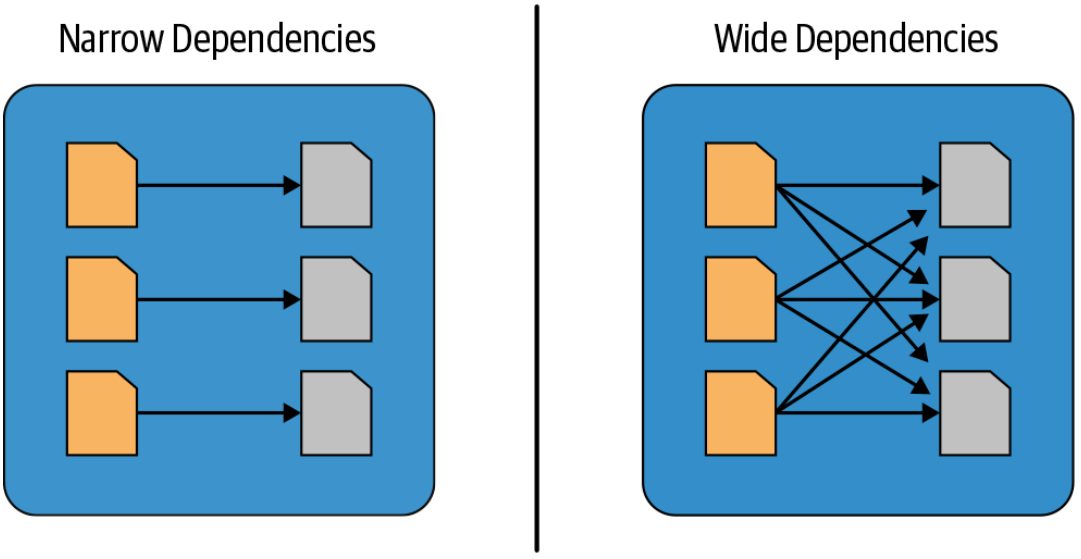

Figure 2-7. Narrow versus wide transformations

The Spark UI

Spark includes a graphical user interface that you can use to inspect or monitor Spark applications in their various stages of decomposition—that is jobs, stages, and tasks. Depending on how Spark is deployed, the driver launches a web UI, running by default on port 4040, where you can view metrics and details such as:

- A list of scheduler stages and tasks
- A summary of RDD sizes and memory usage
- Information about the environment
- Information about the running executors
- All the Spark SQL queries

In local mode, you can access this interface at *http://<localhost>:4040* in a web browser.

NOTE

When you launch `spark-shell`, part of the output shows the localhost URL to access at port 4040.

Let's inspect how the Python example from the previous section translates into jobs, stages, and tasks. To view what the DAG looks like, click on "DAG Visualization" in the web UI. As Figure 2-8 shows, the driver created a single job and a single stage.

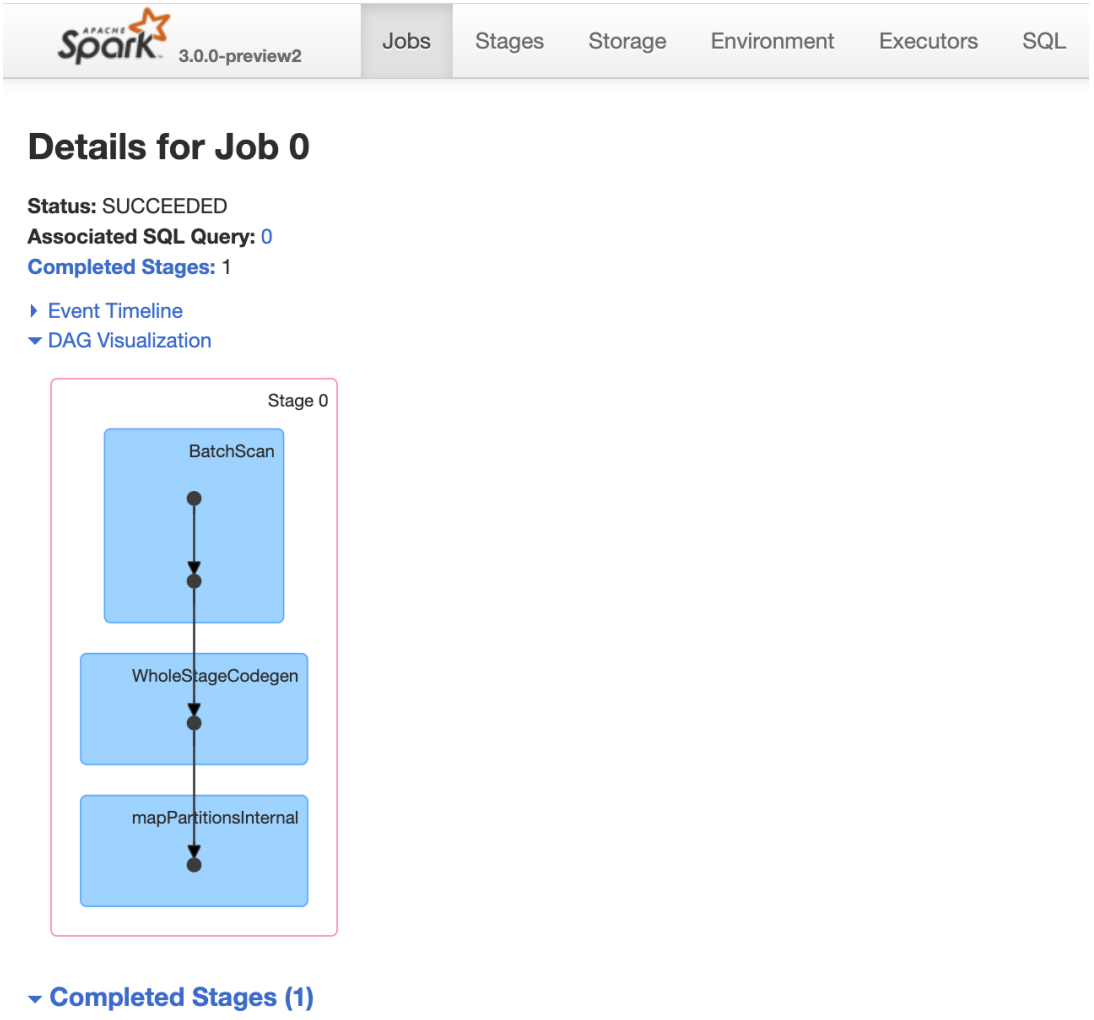

Figure 2-8. The DAG for our simple Python example

Notice that there is no `Exchange`, where data is exchanged between executors, required because there is only a single stage. The individual operations of the stage are shown in blue boxes.

Stage 0 is comprised of one task. If you have multiple tasks, they will be executed in parallel. You can view the details of each stage in the Stages tab, as shown in Figure 2-9.

Figure 2-9. Details of stage 0

We will cover the Spark UI in more detail in Chapter 7. For now, just note that the UI provides a microscopic lens into Spark's internal workings as a tool for debugging and inspecting.

DATABRICKS COMMUNITY EDITION

Databricks is a company that offers a managed Apache Spark platform in the cloud. Aside from using your local machine to run Spark in local mode, you can try some of the examples in this and other chapters using the free Databricks Community Edition (Figure 2-10). As a learning tool for Apache Spark, the Community Edition has many tutorials and examples worthy of note. As well as writing your own notebooks in Python, R, Scala, or SQL, you can also import other notebooks, including Jupyter notebooks.

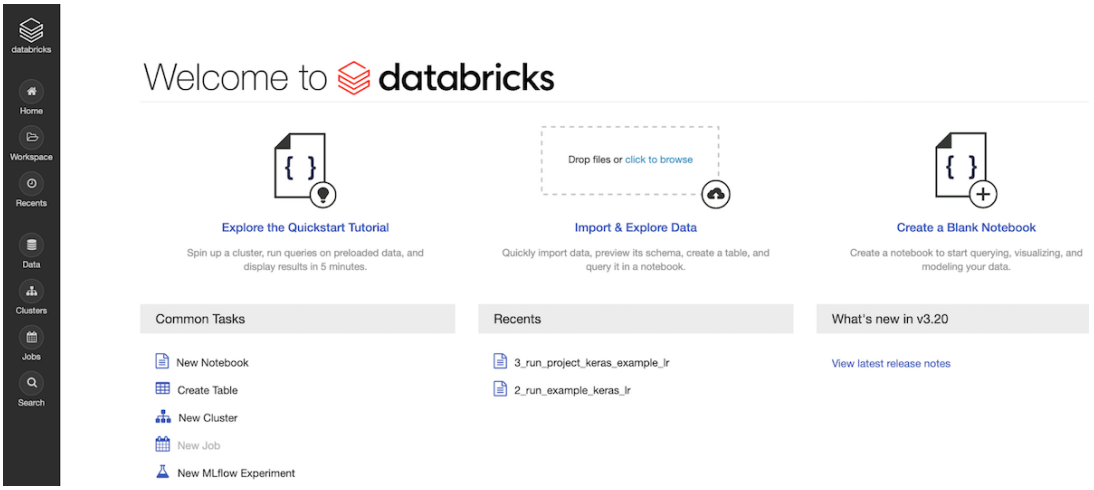

Figure 2-10. Databricks Community Edition

To get an account, go to *https://www.databricks.com/try-databricks* and follow the instructions to try the Community Edition for free. Once registered, you can import the notebooks for this book from its GitHub repo.

Your First Standalone Application

To facilitate learning and exploring, the Spark distribution comes with a set of sample applications for each of Spark's components. You are welcome to peruse the *examples* directory in your installation location to get an idea of what's available.

From the installation directory on your local machine, you can run one of the several Java or Scala sample programs that are provided using the command `bin/run-example <class> [params]`. For example:

```
$ ./bin/run-example JavaWordCount README.md
```

This will spew out `INFO` messages on your console along with a list of each word in the *README.md* file and its count (counting words is the "Hello, World" of distributed computing).

Counting M&Ms for the Cookie Monster

In the previous example, we counted words in a file. If the file were huge, it would be distributed across a cluster partitioned into small chunks of data, and our Spark program would distribute the task of counting each word in each partition and return us the final aggregated count. But that example has become a bit of a cliche.

Let's solve a similar problem, but with a larger data set and using more of Spark's distribution functionality and DataFrame APIs. We will cover the APIs used in this program in later chapters, but for now bear with us.

Among the authors of this book is a data scientist who loves to bake cookies with M&Ms in them, and she rewards her students in the US states where she frequently teaches machine learning and data science courses with batches of those cookies. But she's data-driven, obviously, and wants to ensure that she gets the right colors of M&Ms in the cookies for students in the different states (Figure 2-11).

Figure 2-11. Distribution of M&Ms by color (source: *https://oreil.ly/mhWIT*)

Let's write a Spark program that reads a file with over 100,000 entries (where each row or line has a `<state, mnm_color, count>`) and computes and aggregates the counts for each color and state. These aggregated counts tell us the colors of M&Ms favored by students in each state. The complete Python listing is provided in Example 2-1.

Example 2-1. Counting and aggregating M&Ms (Python version)

```
# Import the necessary libraries.
# Since we are using Python, import the SparkSession and related functions
# from the PySpark module.
import sys

from pyspark.sql import SparkSession

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: mnmcount <file>", file=sys.stderr)
        sys.exit(-1)

    # Build a SparkSession using the SparkSession APIs.
    # If one does not exist, then create an instance. There
    # can only be one SparkSession per JVM.
    spark = (SparkSession
      .builder
      .appName("PythonMnMCount")
      .getOrCreate())
    # Get the M&M data set filename from the command-line arguments
    mnm_file = sys.argv[1]
    # Read the file into a Spark DataFrame using the CSV
    # format by inferring the schema and specifying that the
    # file contains a header, which provides column names for comma-
    # separated fields.
    mnm_df = (spark.read.format("csv")
      .option("header", "true")
      .option("inferSchema", "true")
      .load(mnm_file))

    # We use the DataFrame high-level APIs. Note
    # that we don't use RDDs at all. Because some of Spark's
    # functions return the same object, we can chain function calls.
    # 1. Select from the DataFrame the fields "State", "Color", and "Count"
    # 2. Since we want to group each state and its M&M color count,
```

```
#    we use groupBy()
# 3. Aggregate counts of all colors and groupBy() State and Color
# 4  orderBy() in descending order
count_mnm_df = (mnm_df
  .select("State", "Color", "Count")
  .groupBy("State", "Color")
  .sum("Count")
  .orderBy("sum(Count)", ascending=False))
# Show the resulting aggregations for all the states and colors;
# a total count of each color per state.
# Note show() is an action, which will trigger the above
# query to be executed.
count_mnm_df.show(n=60, truncate=False)
print("Total Rows = %d" % (count_mnm_df.count()))
# While the above code aggregated and counted for all
# the states, what if we just want to see the data for
# a single state, e.g., CA?
# 1. Select from all rows in the DataFrame
# 2. Filter only CA state
# 3. groupBy() State and Color as we did above
# 4. Aggregate the counts for each color
# 5. orderBy() in descending order
# Find the aggregate count for California by filtering
ca_count_mnm_df = (mnm_df
  .select("State", "Color", "Count")
  .where(mnm_df.State == "CA")
  .groupBy("State", "Color")
  .sum("Count")
  .orderBy("sum(Count)", ascending=False))
# Show the resulting aggregation for California.
# As above, show() is an action that will trigger the execution of the
# entire computation.
ca_count_mnm_df.show(n=10, truncate=False)
# Stop the SparkSession
spark.stop()
```

You can enter this code into a Python file called *mnmcount.py* using your favorite editor, download the *mnn_dataset.csv* file from this book's GitHub repo, and submit it as a Spark job using the `submit-spark` script in the installation's *bin* directory. Set your `SPARK_HOME` environment variable to the root-level directory where you installed Spark on your local machine.

NOTE

The preceding code uses the DataFrame API, which reads like high-level DSL queries. We will cover this and the other APIs in the next chapter; for now, note the clarity and simplicity with which you can instruct Spark what to do, not how to do it, unlike with the RDD API. Cool stuff!

To avoid having verbose `INFO` messages printed to the console, copy the *log4j.properties.template* file to *log4j.properties* and set `log4j.rootCategory=WARN` in the *conf/log4j.properties* file.

Let's submit our first Spark job using the Python APIs (for an explanation of what the code does, please read the inline comments in Example 2-1):

```
$SPARK_HOME/bin/spark-submit mnmcount.py data/mnm_dataset.csv

-----+------+----------+
|State| Color|sum(Count)|
+-----+------+----------+
|   CA|Yellow|    100956|
|   WA| Green|     96486|
|   CA| Brown|     95762|
|   TX| Green|     95753|
|   TX|   Red|     95404|
|   CO|Yellow|     95038|
|   NM|   Red|     94699|
|   OR|Orange|     94514|
|   WY| Green|     94339|
|   NV|Orange|     93929|
|   TX|Yellow|     93819|
|   CO| Green|     93724|
|   CO| Brown|     93692|
|   CA| Green|     93505|
|   NM| Brown|     93447|
|   CO|  Blue|     93412|
|   WA|   Red|     93332|
|   WA| Brown|     93082|
|   WA|Yellow|     92920|
|   NM|Yellow|     92747|
|   NV| Brown|     92478|
|   TX|Orange|     92315|
|   AZ| Brown|     92287|
```

```
|   AZ| Green|     91882|
|   WY|   Red|     91768|
|   AZ|Orange|     91684|
|   CA|   Red|     91527|
|   WA|Orange|     91521|
|   NV|Yellow|     91390|
|   UT|Orange|     91341|
|   NV| Green|     91331|
|   NM|Orange|     91251|
|   NM| Green|     91160|
|   WY|  Blue|     91002|
|   UT|   Red|     90995|
|   CO|Orange|     90971|
|   AZ|Yellow|     90946|
|   TX| Brown|     90736|
|   OR|  Blue|     90526|
|   CA|Orange|     90311|
|   OR|   Red|     90286|
|   NM|  Blue|     90150|
|   AZ|   Red|     90042|
|   NV|  Blue|     90003|
|   UT|  Blue|     89977|
|   AZ|  Blue|     89971|
|   WA|  Blue|     89886|
|   OR| Green|     89578|
|   CO|   Red|     89465|
|   NV|   Red|     89346|
|   UT|Yellow|     89264|
|   OR| Brown|     89136|
|   CA|  Blue|     89123|
|   UT| Brown|     88973|
|   TX|  Blue|     88466|
|   UT| Green|     88392|
|   OR|Yellow|     88129|
|   WY|Orange|     87956|
|   WY|Yellow|     87800|
|   WY| Brown|     86110|
+-----+------+----------+

Total Rows = 60

+-----+------+----------+
|State| Color|sum(Count)|
+-----+------+----------+
|   CA|Yellow|    100956|
```

```
|   CA| Brown|     95762|
|   CA| Green|     93505|
|   CA|   Red|     91527|
|   CA|Orange|     90311|
|   CA|  Blue|     89123|
+-----+------+----------+
```

First we see all the aggregations for each M&M color for each state, followed by those only for CA (where the preferred color is yellow).

What if you want to use a Scala version of this same Spark program? The APIs are similar; in Spark, parity is well preserved across the supported languages, with minor syntax differences. Example 2-2 is the Scala version of the program. Take a look, and in the next section we'll show you how to build and run the application.

Example 2-2. Counting and aggregating M&Ms (Scala version)

```
package main.scala.chapter2

import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

/**
 * Usage: MnMcount <mnm_file_dataset>
 */
object MnMcount {
 def main(args: Array[String]) {
   val spark = SparkSession
     .builder
     .appName("MnMCount")
     .getOrCreate()

   if (args.length < 1) {
     print("Usage: MnMcount <mnm_file_dataset>")
     sys.exit(1)
   }
   // Get the M&M data set filename
   val mnmFile = args(0)
   // Read the file into a Spark DataFrame
   val mnmDF = spark.read.format("csv")
     .option("header", "true")
     .option("inferSchema", "true")
```

```
      .load(mnmFile)
    // Aggregate counts of all colors and groupBy() State and Color
    // orderBy() in descending order
    val countMnMDF = mnmDF
      .select("State", "Color", "Count")
      .groupBy("State", "Color")
      .sum("Count")
      .orderBy(desc("sum(Count)"))
    // Show the resulting aggregations for all the states and colors
    countMnMDF.show(60)
    println(s"Total Rows = ${countMnMDF.count()}")
    println()
    // Find the aggregate counts for California by filtering
    val caCountMnNDF = mnmDF
      .select("State", "Color", "Count")
      .where(col("State") === "CA")
      .groupBy("State", "Color")
      .sum("Count")
      .orderBy(desc("sum(Count)"))
    // Show the resulting aggregations for California
    caCountMnMDF.show(10)
    // Stop the SparkSession
    spark.stop()
  }
}
```

Building Standalone Applications in Scala

We will now show you how to build your first Scala Spark program, using the Scala Build Tool (sbt).

NOTE

Because Python is an interpreted language and there is no such step as compiling first (though it's possible to compile your Python code into bytecode in *.pyc*), we will not go into this step here. For details on how to use Maven to build Java Spark programs, we refer you to the guide on the Apache Spark website. For brevity in this book, we cover examples mainly in Python and Scala.

---

build.sbt is the specification file that, like a makefile, describes and instructs the Scala compiler to build your Scala-related tasks, such as jars,

packages, what dependencies to resolve, and where to look for them. In our case, we have a simple sbt file for our M&M code (Example 2-3).

Example 2-3. sbt build file

```
// Name of the package
name := "main/scala/chapter2"
// Version of our package
version := "1.0"
// Version of Scala
scalaVersion := "2.12.10"
// Spark library dependencies
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "3.0.0-preview2",
  "org.apache.spark" %% "spark-sql"  % "3.0.0-preview2"
)
```

Assuming that you have the Java Development Kit (JDK) and sbt installed and `JAVA_HOME` and `SPARK_HOME` set, with a single command, you can build your Spark application:

```
$ sbt clean package
[info] Updated file /Users/julesdamji/gits/LearningSparkV2/chapter2/scala/
project/build.properties: set sbt.version to 1.2.8
[info] Loading project definition from /Users/julesdamji/gits/LearningSparkV2/
chapter2/scala/project
[info] Updating
[info] Done updating.
...
[info] Compiling 1 Scala source to /Users/julesdamji/gits/LearningSparkV2/
chapter2/scala/target/scala-2.12/classes ...
[info] Done compiling.
[info] Packaging /Users/julesdamji/gits/LearningSparkV2/chapter2/scala/target/
scala-2.12/main-scala-chapter2_2.12-1.0.jar ...
[info] Done packaging.
[success] Total time: 6 s, completed Jan 11, 2020, 4:11:02 PM
```

After a successful build, you can run the Scala version of the M&M count example as follows:

```
$SPARK_HOME/bin/spark-submit --class main.scala.chapter2.MnMcount \
jars/main-scala-chapter2_2.12-1.0.jar data/mnm_dataset.csv
...
...
20/01/11 16:00:48 INFO TaskSchedulerImpl: Killing all running tasks in stage 4:
Stage finished
20/01/11 16:00:48 INFO DAGScheduler: Job 4 finished: show at MnMcount.scala:49,
took 0.264579 s
+-----+------+-----+
|State| Color|Total|
+-----+------+-----+
|   CA|Yellow| 1807|
|   CA| Green| 1723|
|   CA| Brown| 1718|
|   CA|Orange| 1657|
|   CA|   Red| 1656|
|   CA|  Blue| 1603|
+-----+------+-----+
```

The output is the same as for the Python run. Try it!

There you have it—our data scientist author will be more than happy to use this data to decide what colors of M&Ms to use in the cookies she bakes for her classes in any of the states she teaches in.

Summary

In this chapter, we covered the three simple steps you need to take to get started with Apache Spark: downloading the framework, familiarizing yourself with the Scala or PySpark interactive shell, and getting to grips with high-level Spark application concepts and terms. We gave a quick overview of the process by which you can use transformations and actions to write a Spark application, and we briefly introduced using the Spark UI to examine the jobs, stages, and tasks created.

Finally, through a short example, we showed you how you can use the high-level Structured APIs to tell Spark what to do—which brings us to the next chapter, where we examine those APIs in more detail.