Data Engineers Guide to Apache Spark and Delta Lake

Table of Contents

Apache Spark™ has seen immense growth over the past several years, including its compatibility with Delta Lake.

Delta Lake is an open-source storage layer that sits on top of your existing data lake file storage, such as AWS S3, Azure Data Lake Storage, or HDFS. Delta Lake brings reliability, performance, and lifecycle management to data lakes. Databricks is proud to share excerpts from the Delta Lake Quickstart and the book, ***Spark: The Definitive Guide***.

CHAPTER 1: A Gentle Introduction to Spark

Now that we took our history lesson on Apache Spark, it's time to start using it and applying it! This chapter will present a gentle introduction to Spark — we will walk through the core architecture of a cluster, Spark Application, and Spark's Structured APIs using DataFrames and SQL. Along the way we will touch on Spark's core terminology and concepts so that you are empowered start using Spark right away. Let's get started with some basic background terminology and concepts.

Spark's Basic Architecture

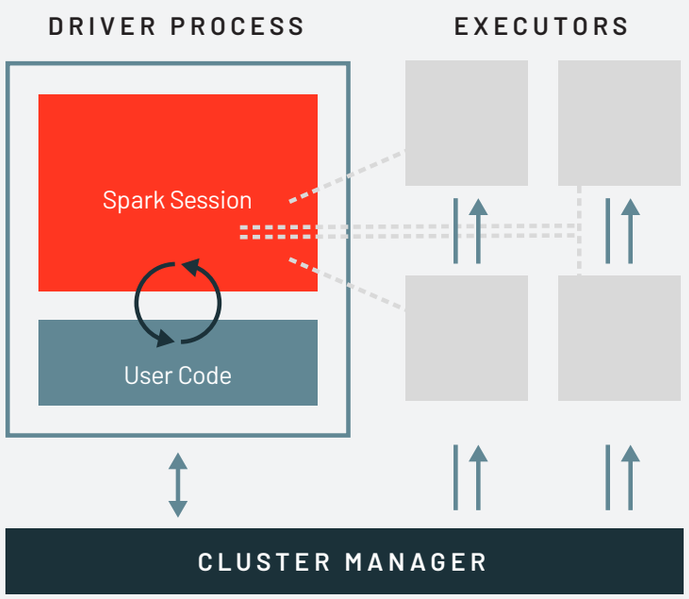
Typically when you think of a "computer" you think about one machine sitting on your desk at home or at work. This machine works perfectly well for watching movies or working with spreadsheet software. However, as many users likely experience at some point, there are some things that your computer is not powerful enough to perform. One particularly challenging area is data processing. Single machines do not have enough power and resources to perform computations on huge amounts of information (or the user may not have time to wait for the computation to finish). A cluster, or group of machines, pools the resources of many machines together allowing us to use all the cumulative resources as if they were one. Now a group of machines alone is not powerful, you need a framework to coordinate work across them. Spark is a tool for just that, managing and coordinating the execution of tasks on data across a cluster of computers.

The cluster of machines that Spark will leverage to execute tasks will be managed by a cluster manager like Spark's Standalone cluster manager, YARN, or Mesos. We then submit Spark Applications to these cluster managers which will grant resources to our application so that we can complete our work.

Spark Applications

Spark Applications consist of a *driver* process and a set of *executor* processes. The driver process runs your `main()` function, sits on a node in the cluster, and is responsible for three things: maintaining information about the Spark Application; responding to a user's program or input; and analyzing, distributing, and scheduling work across the executors (defined momentarily). The driver process is absolutely essential — it's the heart of a Spark Application and maintains all relevant information during the lifetime of the application.

The *executors* are responsible for actually executing the work that the driver assigns them. This means, each executor is responsible for only two things: executing code assigned to it by the driver and reporting the state of the computation, on that executor, back to the driver node.



The cluster manager controls physical machines and allocates resources to Spark Applications. This can be one of several core cluster managers: Spark's standalone cluster manager, YARN, or Mesos. This means that there can be multiple Spark Applications running on a cluster at the same time. We will talk more in depth about cluster managers in ***Part IV: Production Applications*** of this book.

In the previous illustration we see on the left, our driver and on the right the four executors on the right. In this diagram, we removed the concept of cluster nodes. The user can specify how many executors should fall on each node through configurations.

NOTE | *Spark, in addition to its cluster mode, also has a local mode. The driver and executors are simply processes, this means that they can live on the same machine or different machines. In local mode, these both run (as threads) on your individual computer instead of a cluster. We wrote this book with local mode in mind, so everything should be runnable on a single machine.*

As a short review of Spark Applications, the key points to understand at this point are that:

- Spark has some cluster manager that maintains an understanding of the resources available.
- The driver process is responsible for executing our driver program's commands across the executors in order to complete our task.

Now while our executors, for the most part, will always be running Spark code. Our driver can be "driven" from a number of different languages through Spark's Language APIs.

Spark's Language APIs

Spark's language APIs allow you to run Spark code from other languages. For the most part, Spark presents some core "concepts" in every language and these concepts are translated into Spark code that runs on the cluster of machines. If you use the Structured APIs (Part II of this book), you can expect all languages to have the same performance characteristics.

NOTE | *This is a bit more nuanced than we are letting on at this point but for now, it's the right amount of information for new users. In Part II of this book, we'll dive into the details of how this actually works.*

SCALA

Spark is primarily written in Scala, making it Spark's "default" language. This book will include Scala code examples wherever relevant.

JAVA

Even though Spark is written in Scala, Spark's authors have been careful to ensure that you can write Spark code in Java. This book will focus primarily on Scala but will provide Java examples where relevant.

PYTHON

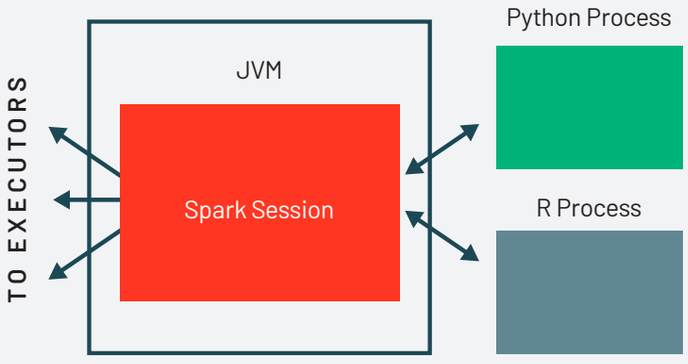
Python supports nearly all constructs that Scala supports. This book will include Python code examples whenever we include Scala code examples and a Python API exists.

SQL

Spark supports ANSI SQL 2003 standard. This makes it easy for analysts and non-programmers to leverage the big data powers of Spark. This book will include SQL code examples wherever relevant

R

Spark has two commonly used R libraries, one as a part of Spark core (SparkR) and another as an R community driven package (sparklyr). We will cover these two different integrations in Part VII: Ecosystem.



Here's a simple illustration of this relationship.

Each language API will maintain the same core concepts that we described above. There is a SparkSession available to the user, the SparkSession will be the entrance point to running Spark code. When using Spark from a Python or R, the user never writes explicit JVM instructions, but instead writes Python and R code that Spark will translate into code that Spark can then run on the executor JVMs.

Spark's APIs

While Spark is available from a variety of languages, what Spark makes available in those languages is worth mentioning. Spark has two fundamental sets of APIs: the low level "Unstructured" APIs and the higher level Structured APIs. We discuss both in this book but these introductory chapters will focus primarily on the higher level APIs.

Starting Spark

Thus far we covered the basic concepts of Spark Applications. This has all been conceptual in nature. When we actually go about writing our Spark Application, we are going to need a way to send user commands and data to the Spark Application. We do that with a SparkSession.

NOTE | *To do this we will start Spark's local mode, just like we did in the previous chapter. This means running* `./bin/spark-shell` *to access the Scala console to start an interactive session. You can also start Python console with* `./bin/pyspark`*. This starts an interactive Spark Application. There is also a process for submitting standalone applications to Spark called* `spark-submit` *where you can submit a precompiled application to Spark. We'll show you how to do that in the next chapter.*

When we start Spark in this interactive mode, we implicitly create a SparkSession which manages the Spark Application. When we start it through a job submission, we must go about creating it or accessing it.

The SparkSession

As discussed in the beginning of this chapter, we control our Spark Application through a driver process. This driver process manifests itself to the user as an object called the SparkSession. The SparkSession instance is the way Spark executes user-defined manipulations across the cluster. There is a one to one correspondence between a SparkSession and a Spark Application. In Scala and Python the variable is available as `spark` when you start up the console. Let's go ahead and look at the `SparkSession` in both Scala and/or Python.

```
spark
```

In **Scala**, you should see something like:

```
res0: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@27159a24
```

In **Python** you'll see something like:

```
<pyspark.sql.session.SparkSession at 0x7efda4c1ccd0>
```

Let's now perform the simple task of creating a range of numbers. This range of numbers is just like a named column in a spreadsheet.

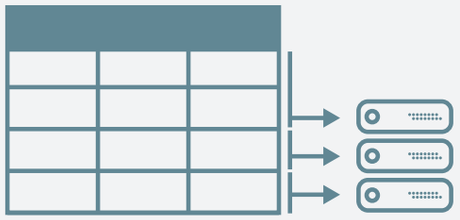
```
%scala
val myRange = spark.range(1000).toDF("number")
```

```
%python
myRange = spark.range(1000).toDF("number")
```

You just ran your first Spark code! We created a DataFrame with one column containing 1000 rows with values from 0 to 999. This range of number represents a *distributed collection*. When run on a cluster, each part of this range of numbers exists on a different executor. This is a Spark DataFrame.

Spreadsheet on a
single machine

Table or DataFrame partitioned
across servers in data center



DataFrames

A *DataFrame* is the most common Structured API and simply represents a table of data with rows and columns. The list of columns and the types in those columns the *schema*. A simple analogy would be a spreadsheet with named columns. The fundamental difference is that while a spreadsheet sits on one computer in one specific location, a Spark DataFrame can span thousands of computers. The reason for putting the data on more than one computer should be intuitive: either the data is too large to fit on one machine or it would simply take too long to perform that computation on one machine.

The DataFrame concept is not unique to Spark. R and Python both have similar concepts. However, Python/R DataFrames (with some exceptions) exist on one machine rather than multiple machines. This limits what you can do with a given DataFrame in python and R to the resources that exist on that specific machine. However, since Spark has language interfaces for both Python and R, it's quite easy to convert to Pandas (Python) DataFrames to Spark DataFrames and R DataFrames to Spark DataFrames (in R).

NOTE | *Spark has several core abstractions: Datasets, DataFrames, SQL Tables, and Resilient Distributed Datasets (RDDs). These abstractions all represent distributed collections of data however they have different interfaces for working with that data. The easiest and most efficient are DataFrames, which are available in all languages. We cover Datasets at the end of Part II and RDDs in Part III of this book. The following concepts apply to all of the core abstractions.*

Partitions

In order to allow every executor to perform work in parallel, Spark breaks up the data into chunks, called partitions. A *partition* is a collection of rows that sit on one physical machine in our cluster. A DataFrame's partitions represent how the data is physically distributed across your cluster of machines during execution. If you have one partition, Spark will only have a parallelism of one even if you have thousands of executors. If you have many partitions, but only one executor Spark will still only have a parallelism of one because there is only one computation resource.

An important thing to note, is that with DataFrames, we do not (for the most part) manipulate partitions manually (on an individual basis). We simply specify high-level transformations of data in the physical partitions and Spark determines how this work will actually execute on the cluster. Lower level APIs do exist (via the Resilient Distributed Datasets interface) and we cover those in Part III of this book.

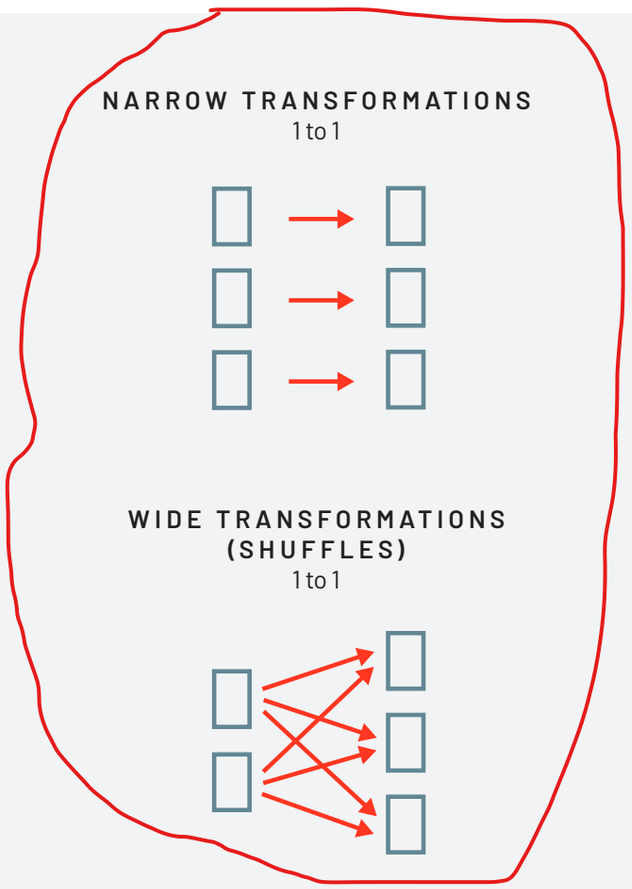
Transformations

In Spark, the core data structures are *immutable* meaning they cannot be changed once created. This might seem like a strange concept at first, if you cannot change it, how are you supposed to use it? In order to "change" a DataFrame you will have to instruct Spark how you would like to modify the DataFrame you have into the one that you want. These instructions are called *transformations*. Let's perform a simple transformation to find all even numbers in our current DataFrame.

```
%scala
val divisBy2 = myRange.where("number % 2 = 0")
```

```
%python
divisBy2 = myRange.where("number % 2 = 0")
```

You will notice that these return no output, that's because we only specified an abstract transformation and Spark will not act on transformations until we call an action, discussed shortly. Transformations are the core of how you will be expressing your business logic using Spark. There are two types of transformations, those that specify narrow dependencies and those that specify wide dependencies.



Transformations consisting of *narrow dependencies* (we'll call them narrow transformations) are those where each input partition will contribute to only one output partition. In the preceding code snippet, our `where` statement specifies a narrow dependency, where only one partition contributes to at most one output partition.

A *wide dependency* (or wide transformation) style transformation will have input partitions contributing to many output partitions. You will often hear this referred to as a *shuffle* where Spark will exchange partitions across the cluster. With narrow transformations, Spark will automatically perform an operation called pipelining on narrow dependencies, this means that if we specify multiple filters on DataFrames they'll all be performed in-memory. The same cannot be said for shuffles. When we perform a shuffle, Spark will write the results to disk. You'll see lots of talks about shuffle optimization across the web because it's an important topic but for now all you need to understand are that there are two kinds of transformations.

We now see how transformations are simply ways of specifying different series of data manipulation. This leads us to a topic called lazy evaluation.

Lazy Evaluation

Lazy evaluation means that Spark will wait until the very last moment to execute the graph of computation instructions. In Spark, instead of modifying the data immediately when we express some operation, we build up a *plan* of transformations that we would like to apply to our source data. Spark, by waiting until the last minute to execute the code, will compile this plan from your raw, DataFrame transformations, to an efficient physical plan that will run as efficiently as possible across the cluster. This provides immense benefits to the end user because Spark can optimize the entire data flow from end to end. An example of this is something called "predicate pushdown" on DataFrames. If we build a large Spark job but specify a filter at the end that only requires us to fetch one row from our source data, the most efficient way to execute this is to access the single record that we need. Spark will actually optimize this for us by pushing the filter down automatically.

Actions

Transformations allow us to build up our logical transformation plan. To trigger the computation, we run an *action*. An *action* instructs Spark to compute a result from a series of transformations. The simplest action is `count` which gives us the total number of records in the DataFrame.

```
divisBy2.count()
```

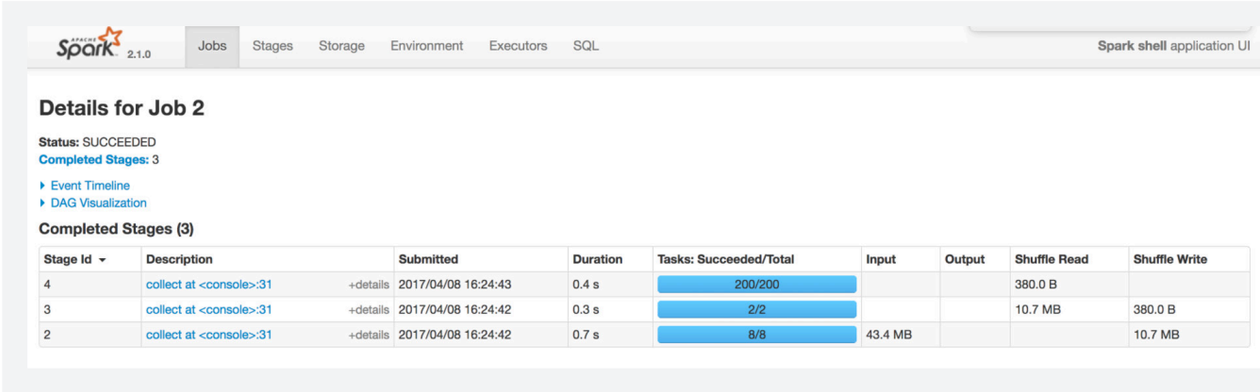
We now see a result! There are 500 numbers divisible by two from 0 to 999 (big surprise!). Now `count` is not the only action. There are three kinds of actions:

- actions to view data in the console;
- actions to collect data to native objects in the respective language;
- and actions to write to output data sources.

In specifying our action, we started a Spark job that runs our filter transformation (a narrow transformation), then an aggregation (a wide transformation) that performs the counts on a per partition basis, then a collect with brings our result to a native object in the respective language. We can see all of this by inspecting the Spark UI, a tool included in Spark that allows us to monitor the Spark jobs running on a cluster.

Spark UI

During Spark's execution of the previous code block, users can monitor the progress of their job through the Spark UI. The Spark UI is available on port 4040 of the driver node. If you are running in local mode this will just be the `http://localhost:4040`. The Spark UI maintains information on the state of our Spark jobs, environment, and cluster state. It's very useful, especially for tuning and debugging. In this case, we can see one Spark job with two stages and nine tasks were executed.



This chapter avoids the details of Spark jobs and the Spark UI, we cover the Spark UI in detail in **Part IV: Production Applications**. At this point you should understand that a Spark job represents a set of transformations triggered by an individual action and we can monitor that from the Spark UI.

An End to End Example

In the previous example, we created a DataFrame of a range of numbers; not exactly groundbreaking big data. In this section we will reinforce everything we learned previously in this chapter with a worked example and explaining step by step what is happening under the hood. We'll be using some flight data available here from the United States Bureau of Transportation statistics.

Inside of the CSV folder linked above, you'll see that we have a number of files. You will also notice a number of other folders with different file formats that we will discuss in **Part II: Reading and Writing Data**. We will focus on the CSV files.

Each file has a number of rows inside of it. Now these files are CSV files, meaning that they're a semi-structured data format with a row in the file representing a row in our future DataFrame.

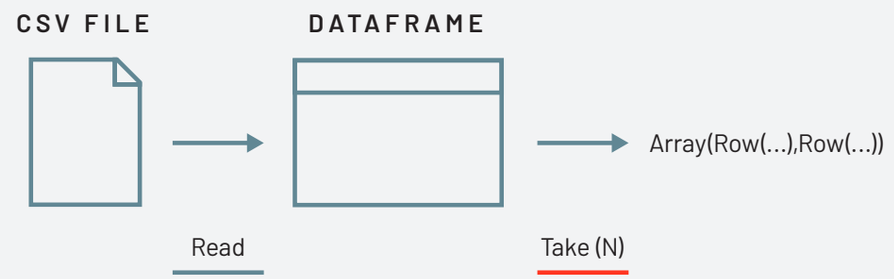
```
$ head /mnt/defg/flight-data/csv/2015-summary.csv
DEST_COUNTRY_NAME,ORIGIN_COUNTRY_NAME,count
United States,Romania,15
United States,Croatia,1
United States,Ireland,344
```

Spark includes the ability to read and write from a large number of data sources. In order to read this data in, we will use a DataFrameReader that is associated with our SparkSession. In doing so, we will specify the file format as well as any options we want to specify. In our case, we want to do something called schema inference, we want Spark to take the best guess at what the schema of our DataFrame should be. The reason for this is that CSV files are not completely structured data formats. We also want to specify that the first row is the header in the file, we'll specify that as an option too.

To get this information Spark will read in a little bit of the data and then attempt to parse the types in those rows according to the types available in Spark. You'll see that this works just fine. We also have the option of strictly specifying a schema when we read in data (which we recommend in production scenarios).

```
%scala
val flightData2015 = spark
  .read
  .option("inferSchema", "true")
  .option("header", "true")
  .csv("/mnt/defg/flight-data/csv/2015-summary.csv")
```

```
%python
flightData2015 = spark\
  .read\
  .option("inferSchema", "true")\
  .option("header", "true")\
  .csv("/mnt/defg/flight-data/csv/2015-summary.csv")
```



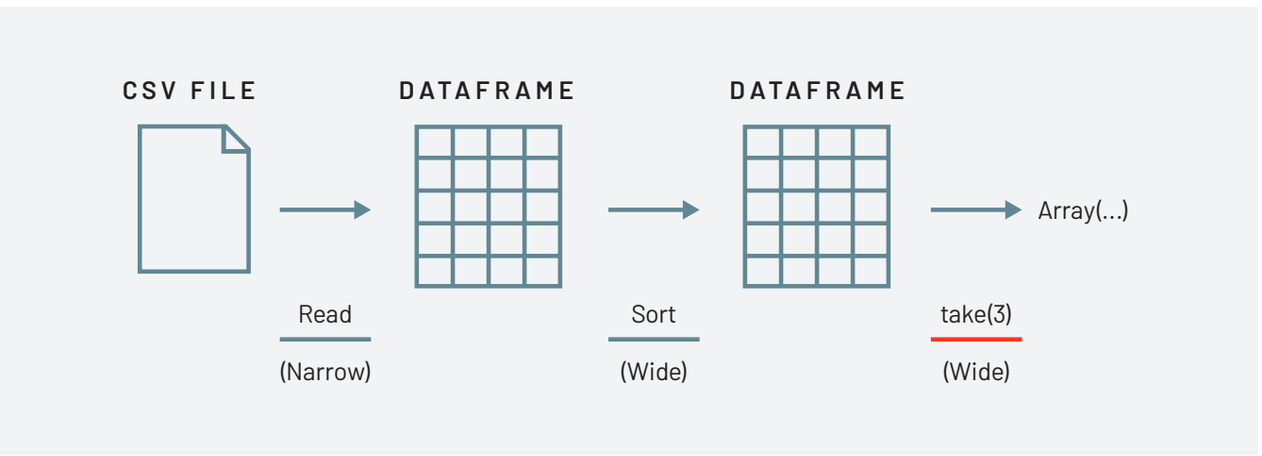
Each of these DataFrames (in Scala and Python) each have a set of columns with an unspecified number of rows. The reason the number of rows is "unspecified" is because reading data is a transformation, and is therefore a lazy operation. Spark only peeked at a couple of rows of data to try to guess what types each column should be.

If we perform the `take` action on the DataFrame, we will be able to see the same results that we saw before when we used the command line.

```
flightData2015.take(3)
Array([United States,Romania,15], [United States,Croatia...
```

Let's specify some more transformations! Now we will sort our data according to the count column which is an integer type.

NOTE | *Remember, the* `sort` *does not modify the DataFrame. We use the sort is a transformation that returns a new DataFrame by transforming the previous DataFrame. Let's illustrate what's happening when we call* `take` *on that resulting DataFrame.*



Nothing happens to the data when we call sort because it's just a transformation. However, we can see that Spark is building up a plan for how it will execute this across the cluster by looking at the `explain` plan. We can call explain on any DataFrame object to see the DataFrame's lineage (or how Spark will execute this query).

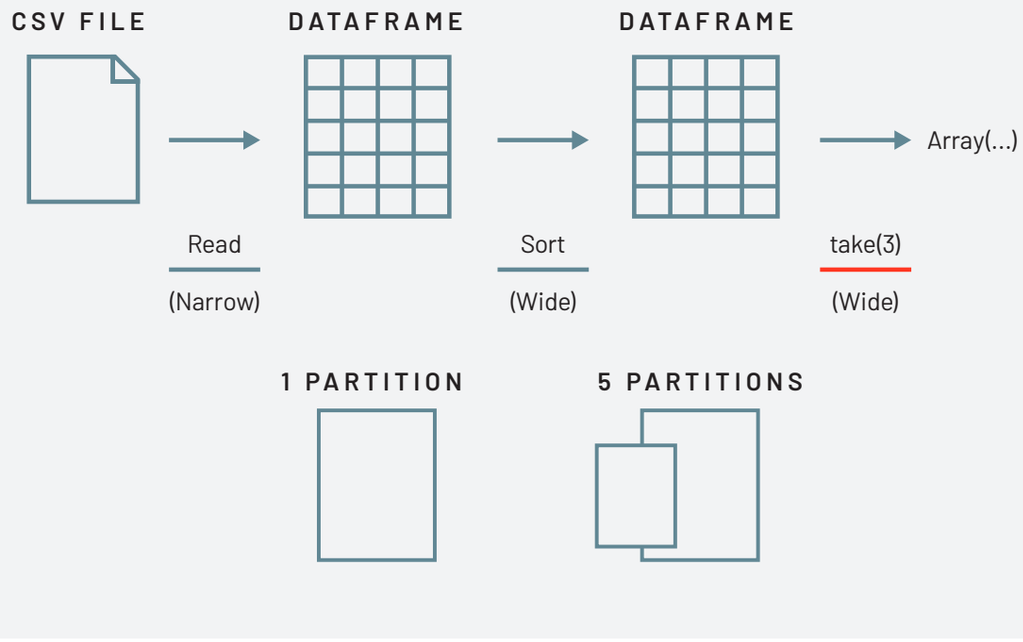
```
flightData2015.sort("count").explain()
```

Congratulations, you've just read your first explain plan! Explain plans are a bit arcane, but with a bit of practice it becomes second nature. Explain plans can be read from top to bottom, the top being the end result and the bottom being the source(s) of data. In our case, just take a look at the first keywords. You will see "sort", "exchange", and "FileScan". That's because the sort of our data is actually a wide transformation because rows will have to be compared with one another. Don't worry too much about understanding everything about explain plans at this point, they can just be helpful tools for debugging and improving your knowledge as you progress with Spark.

Now, just like we did before, we can specify an action in order to kick off this plan. However before doing that, we're going to set a configuration. By default, when we perform a shuffle Spark will output two hundred shuffle partitions. We will set this value to five in order to reduce the number of the output partitions from the shuffle from two hundred to five.

```
spark.conf.set("spark.sql.shuffle.partitions", "5")
flightData2015.sort("count").take(2)
... Array([United States,Singapore,1], [Moldova,United States,1])
```

This operation is illustrated in the following image. You'll notice that in addition to the logical transformations, we include the physical partition count as well.



The logical plan of transformations that we build up defines a lineage for the DataFrame so that at any given point in time Spark knows how to recompute any partition by performing all of the operations it had before on the same input data. This sits at the heart of Spark's programming model, functional programming where the same inputs always result in the same outputs when the transformations on that data stay constant.

We do not manipulate the physical data, but rather configure physical execution characteristics through things like the shuffle partitions parameter we set above. We got five output partitions because that's what we changed the shuffle partition value to. You can change this to help control the physical execution characteristics of your Spark jobs. Go ahead and experiment with different values and see the number of partitions yourself. In experimenting with different values, you should see drastically different run times. Remeber that you can monitor the job progress by navigating to the Spark UI on port 4040 to see the physical and logical execution characteristics of our jobs.

DataFrames and SQL

We worked through a simple example in the previous example, let's now work through a more complex example and follow along in both DataFrames and SQL. Spark the same transformations, regardless of the language, in the exact same way. You can express your business logic in SQL or DataFrames (either in R, Python, Scala, or Java) and Spark will compile that logic down to an underlying plan (that we see in the explain plan) before actually executing your code. Spark SQL allows you as a user to register any DataFrame as a table or view (a temporary table) and query it using pure SQL. There is no performance difference between writing SQL queries or writing DataFrame code, they both "compile" to the same underlying plan that we specify in DataFrame code.

Any DataFrame can be made into a table or view with one simple method call.

```
%scala
flightData2015.createOrReplaceTempView("flight_data_2015")
```

```
%python
flightData2015.createOrReplaceTempView("flight_data_2015")
```

Now we can query our data in SQL. To execute a SQL query, we'll use the `spark.sql` function (remember `spark` is our SparkSession variable?) that conveniently, returns a new DataFrame. While this may seem a bit circular in logic – that a SQL query against a DataFrame returns another DataFrame, it's actually quite powerful. As a user, you can specify transformations in the manner most convenient to you at any given point in time and not have to trade any efficiency to do so! To understand that this is happening, let's take a look at two explain plans.

```
%scala
val sqlWay = spark.sql(“””
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
“””)
val dataFrameWay = flightData2015
  .groupBy(‘DEST_COUNTRY_NAME)
  .count()
sqlWay.explain
dataFrameWay.explain
```

```
%python
sqlWay = spark.sql(“””
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
“””)
dataFrameWay = flightData2015\
.groupBy(“DEST_COUNTRY_NAME”)\
.count()
sqlWay.explain()
dataFrameWay.explain()
```

```
== Physical Plan ==
*HashAggregate(keys=[DEST_COUNTRY_NAME#182], functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#182, 5)
   +- *HashAggregate(keys=[DEST_COUNTRY_NAME#182], functions=[partial_
count(1)])
      +- *FileScan csv [DEST_COUNTRY_NAME#182] ...
== Physical Plan ==
*HashAggregate(keys=[DEST_COUNTRY_NAME#182], functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#182, 5)
   +- *HashAggregate(keys=[DEST_COUNTRY_NAME#182], functions=[partial_
count(1)])
      +- *FileScan csv [DEST_COUNTRY_NAME#182] ...
```

We can see that these plans compile to the exact same underlying plan!

To reinforce the tools available to us, let's pull out some interesting statistics from our data. One thing to understand is that DataFrames (and SQL) in Spark already have a huge number of manipulations available. There are hundreds of functions that you can leverage and import to help you resolve your big data problems faster. We will use the `max` function, to find out what the maximum number of flights to and from any given location are. This just scans each value in relevant column the DataFrame and sees if it's bigger than the previous values that have been seen. This is a transformation, as we are effectively filtering down to one row. Let's see what that looks like.

```
spark.sql("SELECT max(count) from flight_data_2015").take(1)
```

```
%scala
import org.apache.spark.sql.functions.max
flightData2015.select(max("count")).take(1)
```

```
%python
from pyspark.sql.functions import max
flightData2015.select(max("count")).take(1)
```

Great, that's a simple example. Let's perform something a bit more complicated and find out the top five destination countries in the data? This is our first multi-transformation query so we'll take it step by step. We will start with a fairly straightforward SQL aggregation.

```
%scala
val maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")
maxSql.collect()
```

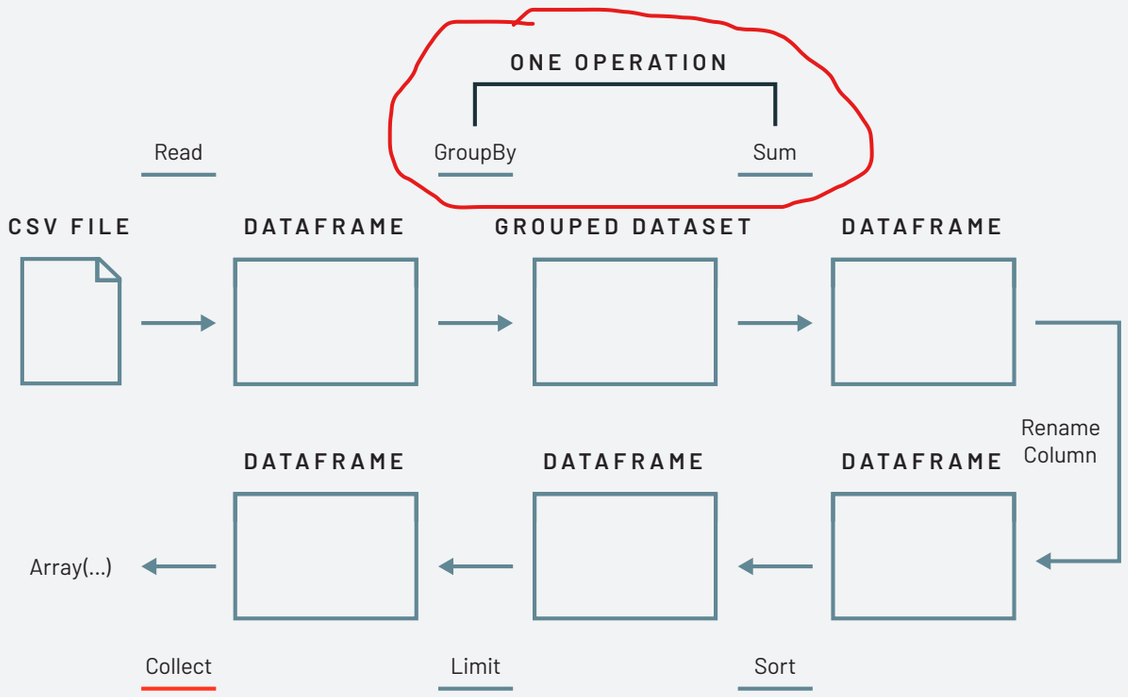
```
%python
maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")
maxSql.collect()
```

Now let's move to the DataFrame syntax that is semantically similar but slightly different in implementation and ordering. But, as we mentioned, the underlying plans for both of them are the same. Let's execute the queries and see their results as a sanity check.

```
%scala
import org.apache.spark.sql.functions.desc
flightData2015
  .groupBy("DEST_COUNTRY_NAME")
  .sum("count")
  .withColumnRenamed("sum(count)", "destination_total")
  .sort(desc("destination_total"))
  .limit(5)
  .collect()
```

```
%python
from pyspark.sql.functions import desc
flightData2015\
  .groupBy("DEST_COUNTRY_NAME")\
  .sum("count")\
  .withColumnRenamed("sum(count)", "destination_total")\
  .sort(desc("destination_total"))\
  .limit(5)\
  .collect()
```

Now there are 7 steps that take us all the way back to the source data. You can see this in the explain plan on those DataFrames. Illustrated below are the set of steps that we perform in "code". The true execution plan (the one visible in explain) will differ from what we have below because of optimizations in physical execution, however the illustration is as good of a starting point as any. This execution plan is a directed *acyclic graph (DAG)* of transformations, each resulting in a new immutable DataFrame, on which we call an action to generate a result.



The first step is to read in the data. We defined the DataFrame previously but, as a reminder, Spark does not actually read it in until an action is called on that DataFrame or one derived from the original DataFrame.

The second step is our grouping, technically when we call `groupBy` we end up with a RelationalGroupedDataset which is a fancy name for a DataFrame that has a grouping specified but needs the user to specify an aggregation before it can be queried further. We can see this by trying to perform an action on it (which will not work). We basically specified that we're going to be grouping by a key (or set of keys) and that now we're going to perform an aggregation over each one of those keys.

Therefore the third step is to specify the aggregation. Let's use the `sum` aggregation method. This takes as input a column expression or simply, a column name. The result of the sum method call is a new dataFrame. You'll see that it has a new schema but that it does know the type of each column. It's important to reinforce (again!) that no computation has been performed. This is simply another transformation that we've expressed and Spark is simply able to trace the type information we have supplied.

The fourth step is a simple renaming, we use the `withColumnRenamed` method that takes two arguments, the original column name and the new column name. Of course, this doesn't perform computation — this is just another transformation!

The fifth step sorts the data such that if we were to take results off of the top of the DataFrame, they would be the largest values found in the `destination_total` column.

You likely noticed that we had to import a function to do this, the `desc` function. You might also notice that `desc` does not return a string but a `Column`. In general, many DataFrame methods will accept Strings (as column names) or `Column` types or expressions. Columns and expressions are actually the exact same thing.

Penultimately, we'll specify a limit. This just specifies that we only want five values. This is just like a filter except that it filters by position instead of by value. It's safe to say that it basically just specifies a `DataFrame` of a certain size.

The last step is our action! Now we actually begin the process of collecting the results of our DataFrame above and Spark will give us back a list or array in the language that we're executing. Now to reinforce all of this, let's look at the explain plan for the above query.

```
%scala
flightData2015
   .groupBy("DEST_COUNTRY_NAME")
   .sum("count")
   .withColumnRenamed("sum(count)", "destination_total")
   .sort(desc("destination_total"))
   .limit(5)
   .explain()
```

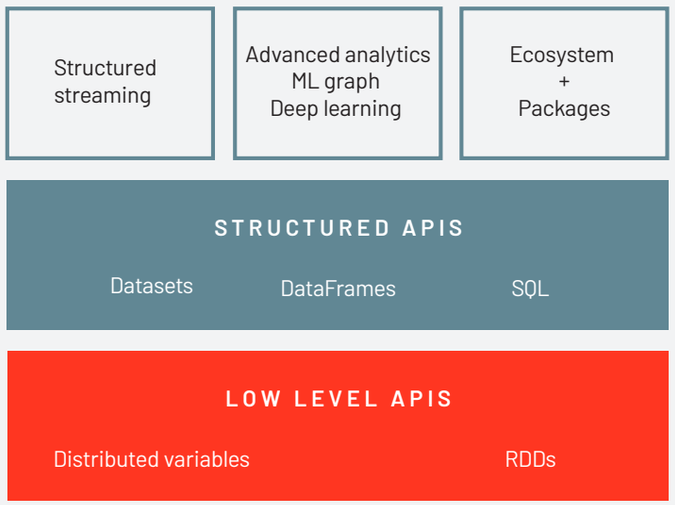
```
%python
flightData2015\
.groupBy("DEST_COUNTRY_NAME")\
.sum("count")\
.withColumnRenamed("sum(count)", "destination_total")\
.sort(desc("destination_total"))\
   .limit(5)\
   .explain()
```

```
== Physical Plan ==
TakeOrderedAndProject(limit=5, orderBy=[destination_total#16194L DESC], output=[DEST_COUNTRY_NAME#7323,...
+- *HashAggregate(keys=[DEST_COUNTRY_NAME#7323], functions=[sum(count#7325L)])
  +- Exchange hashpartitioning(DEST_COUNTRY_NAME#7323, 5)
      +- *HashAggregate(keys=[DEST_COUNTRY_NAME#7323], functions=[partial
sum(count#7325L)])
         +- InMemoryTableScan [DEST_COUNTRY_NAME#7323, count#7325L]
               +- InMemoryRelation [DEST_COUNTRY_NAME#7323, ORIGIN_COUNTRY_NAME#7324, count#7325L]...
                     +- *Scan csv [DEST_COUNTRY_NAME#7578,ORIGIN_COUNTRY_NAME#7579,count#7580L]...
```

While this explain plan doesn't match our exact "conceptual plan" all of the pieces are there. You can see the limit statement as well as the `orderBy` (in the first line). You can also see how our aggregation happens in two phases, in the `partial_sum` calls. This is because summing a list of numbers is commutative and Spark can perform the sum, partition by partition. Of course we can see how we read in the DataFrame as well.

Naturally, we don't always have to collect the data. We can also write it out to any data source that Spark supports. For instance, let's say that we wanted to store the information in a database like PostgreSQL or write them out to another file.

CHAPTER 2: A Tour of Spark's Toolset



In the previous chapter we introduced Spark's core concepts, like transformations and actions, in the context of Spark's Structured APIs. These simple conceptual building blocks are the foundation of Apache Spark's vast ecosystem of tools and libraries. Spark is composed of the simple primitives, the lower level APIs and the Structured APIs, then a series of "standard libraries" included in Spark.

Developers use these tools for a variety of different tasks, from graph analysis and machine learning to streaming and integrations with a host of libraries and databases. This chapter will present a whirlwind tour of much of what Spark has to offer. Each section in this chapter are elaborated upon by other parts of this book, this chapter is simply here to show you what's possible.

This chapter will cover:

- Production applications with spark-submit
- Datasets: structured and type safe APIs
- Structured Streaming
- Machine learning and advanced analytics
- Spark's lower level APIs
- SparkR
- Spark's package ecosystem

The entire book covers these topics in depth, the goal of this chapter is simply to provide a whirlwind tour of Spark. Once you've gotten the tour, you'll be able to jump to many different parts of the book to find answers to your questions about particular topics. This chapter aims for breadth, instead of depth. Let's get started!

Production Applications

Spark makes it easy to make simple to reason about and simple to evolve big data programs. Spark also makes it easy to turn in your interactive exploration into production applications with a tool called `spark-submit` that is included in the core of Spark. spark-submit does one thing, it allows you to submit your applications to a currently managed cluster to run. When you submit this, the application will run until the application exists or errors. You can do this with all of Spark's support cluster managers including Standalone, Mesos, and YARN.

In the process of doing so, you have a number of knobs that you can turn and control to specify the resources this application has as well, how it should be run, and the parameters for your specific application.

You can write these production applications in any of Spark's supported languages and then submit those applications for execution. The simplest example is one that you can do on your local machine by running the following command line snippet on your local machine in the directory into which you downloaded Spark.

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local \
  ./examples/jars/spark-examples_2.11-2.2.0.jar 10
```

What this will do is calculate the digits of pi to a certain level of estimation. What we've done here is specified that we want to run it on our local machine, specified which class and which jar we would like to run as well as any command line arguments to that particular class.

We can do this in Python with the following command line arguments.

```
./bin/spark-submit \
  --master local \
  ./examples/src/main/python/pi.py 10
```

By swapping out the path to the file and the cluster configurations, we can write and run production applications. Now Spark provides a lot more than just DataFrames that we can run as production applications. The rest of this chapter will walk through several different APIs that we can leverage to run all sorts of production applications.

Datasets: Type-Safe Structured APIs

The next topic we'll cover is a type-safe version of Spark's structured API for Java and Scala, called Datasets. This API is not available in Python and R, because those are dynamically typed languages, but it is a powerful tool for writing large applications in Scala and Java.

Recall that DataFrames, which we saw earlier, are a distributed collection of objects of type Row, which can hold various types of tabular data. The Dataset API allows users to assign a Java class to the records inside a DataFrame, and manipulate it as a collection of typed objects, similar to a Java `ArrayList` or Scala `Seq`. The APIs available on Datasets are *type-safe*, meaning that you cannot accidentally view the objects in a Dataset as being of another class than the class you put in initially. This makes Datasets especially attractive for writing large applications where multiple software engineers must interact through well-defined interfaces.

The Dataset class is parametrized with the type of object contained inside: `Dataset<T>` in Java and `Dataset[T]` in Scala. As of Spark 2.0, the types `T` supported are all classes following the JavaBean pattern in Java, and `case classes` in Scala. These types are restricted because Spark needs to be able to automatically analyze the type T and create an appropriate schema for the tabular data inside your Dataset.

The awesome thing about Datasets is that we can use them only when we need or want to. For instance, in the follow example I'll define my own object and manipulate it via arbitrary map and filter functions. Once we've performed our manipulations, Spark can automatically turn it back into a DataFrame and we can manipulate it further using the hundreds of functions that Spark includes. This makes it easy to drop down to lower level, perform type-safe coding when necessary, and move higher up to SQL for more rapid analysis. We cover this material extensively in the next part of this book, but here is a small example showing how we can use both type-safe functions and DataFrame-like SQL expressions to quickly write business logic.

```
%scala
// A Scala case class (similar to a struct) that will automatically
// be mapped into a structured data table in Spark
case class Flight(DEST_COUNTRY_NAME: String, ORIGIN_COUNTRY_NAME: String, count: BigInt)
val flightsDF = spark.read.parquet("/mnt/defg/flight-data/parquet/2010-summary.parquet/")
val flights = flightsDF.as[Flight]
```

One final advantage is that when you call `collect` or `take` on a Dataset, we're going to collect to objects of the proper type in your Dataset, not DataFrame Rows. This makes it easy to get type safety and safely perform manipulation in a distributed and a local manner without code changes.

```
%scala
flights
  .filter(flight_row => flight_row.ORIGIN_COUNTRY_NAME != "Canada")
  .take(5)
```

Structured Streaming

Structured Streaming is a high-level API for stream processing that became production-ready in Spark 2.2. Structured Streaming allows you to take the same operations that you perform in batch mode using Spark's structured APIs, and run them in a streaming fashion. This can reduce latency and allow for incremental processing. The best thing about Structured Streaming is that it allows you to rapidly and quickly get value out of streaming systems with virtually no code changes. It also makes it easy to reason about because you can write your batch job as a way to prototype it and then you can convert it to streaming job. The way all of this works is by incrementally processing that data.

Let's walk through a simple example of how easy it is to get started with Structured Streaming. For this we will use a retail dataset. One that has specific dates and times for us to be able to use. We will use the "by-day" set of files where one file represents one day of data.

We put it in this format to simulate data being produced in a consistent and regular manner by a different process. Now this is retail data so imagine that these are being produced by retail stores and sent to a location where they will be read by our Structured Streaming job.

It's worth sharing a sample of the data so you can reference what the data looks like.

```
InvoiceNo,StockCode,Description,Quantity,InvoiceDate,UnitPrice,CustomerID,Country
536365,85123A,WHITE HANGING HEART T-LIGHT HOLDER,6,2010-12-01 08:26:00,2.55,17850.0,United Kingdom
536365,71053,WHITE METAL LANTERN,6,2010-12-01 08:26:00,3.39,17850.0,United Kingdom
536365,84406B,CREAM CUPID HEARTS COAT HANGER,8,2010-12-01 08:26:00,2.75,17850.0,United Kingdom
```

Now in order to ground this, let's first analyze the data as a static dataset and create a DataFrame to do so. We'll also create a schema from this static dataset. There are ways of using schema inference with streaming that we will touch on in the Part V of this book.

```
%scala
val staticDataFrame = spark.read.format("csv")
   .option("header", "true")
   .option("inferSchema", "true")
   .load("/mnt/defg/retail-data/by-day/*.csv")
staticDataFrame.createOrReplaceTempView("retail_data")
val staticSchema = staticDataFrame.schema
```

```
%python
staticDataFrame = spark.read.format("csv")\
   .option("header", "true")\
   .option("inferSchema", "true")\
   .load("/mnt/defg/retail-data/by-day/*.csv")
staticDataFrame.createOrReplaceTempView("retail_data")
staticSchema = staticDataFrame.schema
```

Now since we're working with time series data it's worth mentioning how we might go along grouping and aggregating our data. In this example we'll take a look at the largest sale hours where a given customer (identified by `CustomerId`) makes a large purchase. For example, let's add a total cost column and see on what days a customer spent the most.

The window function will include all data from each day in the aggregation. It's simply a window over the time series column in our data. This is a helpful tool for manipulating date and timestamps because we can specify our requirements in a more human form (via intervals) and Spark will group all of them together for us.

```
%scala
import org.apache.spark.sql.functions.{window, column, desc, col}
staticDataFrame
  .selectExpr(
  “CustomerId”,
  “(UnitPrice * Quantity) as total_cost”,
  “InvoiceDate”)
  .groupBy(
   col(“CustomerId”), window(col(“InvoiceDate”), “1 day”))
  .sum(“total_cost”)
  .show(5)
```

```
%python
from pyspark.sql.functions import window, column, desc, col
staticDataFrame\
  .selectExpr(
  “CustomerId”,
  “(UnitPrice * Quantity) as total_cost” ,
  “InvoiceDate” )\
  .groupBy(
   col(“CustomerId”), window(col(“InvoiceDate”), “1 day”))\
  .sum(“total_cost”)\
  .show(5)
```

It’s worth mentioning that we can also run this as SQL code, just as we saw in the previous chapter.

Here’s a sample of the output that you’ll see.

```
+----------+--------------------+------------------+
|CustomerId|              window|   sum(total_cost)|
+----------+--------------------+------------------+
|   17450.0|[2011-09-20 00:00...|          71601.44|
|      null|[2011-11-14 00:00...|          55316.08|
|      null|[2011-11-07 00:00...|          42939.17|
|      null|[2011-03-29 00:00...| 33521.39999999998|
|      null|[2011-12-08 00:00...|31975.590000000007|
+----------+--------------------+------------------+
```

The null values represent the fact that we don’t have a customerId for some transactions.

That’s the static DataFrame version, there shouldn’t be any big surprises in there if you’re familiar with the syntax. Now we’ve seen how that works, let’s take a look at the streaming code! You’ll notice that very little actually changes about our code. The biggest change is that we used `readStream` instead of `read`, additionally you’ll notice `maxFilesPerTrigger` option which simply specifies the number of files we should read in at once. This is to make our demonstration more “streaming” and in a production scenario this would be omitted.

Now since you're likely running this in local mode, it's a good practice to set the number of shuffle partitions to something that's going to be a better fit for local mode. This configuration simple specifies the number of partitions that should be created after a shuffle, by default the value is two hundred but since there aren't many executors on this machine it's worth reducing this to five. We did this same operation in the previous chapter, so if you don't remember why this is important feel free to flip back to the previous chapter to review.

```
%scala
val streamingDataFrame = spark.readStream
   .schema(staticSchema)
   .option("maxFilesPerTrigger", 1)
   .format("csv")
   .option("header", "true")
   .load("d/mnt/defg/retail-data/by-day/*.csv")
```

```
%python
streamingDataFrame = spark.readStream\
   .schema(staticSchema)\
   .option("maxFilesPerTrigger", 1)\
   .format("csv")\
   .option("header", "true")\
   .load("/mnt/defg/retail-data/by-day/*.csv")
```

Now we can see the DataFrame is streaming.

```
streamingDataFrame.isStreaming // returns true
```

Let's set up the same business logic as the previous DataFrame manipulation, we'll perform a summation in the process.

```
%scala
val purchaseByCustomerPerHour = streamingDataFrame
   .selectExpr(
   "CustomerId",
   "(UnitPrice * Quantity) as total_cost",
   "InvoiceDate")
   .groupBy(
    $"CustomerId", window($"InvoiceDate", "1 day"))
   .sum("total_cost")
```

```
%python
purchaseByCustomerPerHour = streamingDataFrame\
   .selectExpr(
   "CustomerId",
   "(UnitPrice * Quantity) as total_cost" ,
   "InvoiceDate" )\
   .groupBy(
    col("CustomerId"), window(col("InvoiceDate"), "1 day"))\
   .sum("total_cost")
```

This is still a lazy operation, so we will need to call a streaming action to start the execution of this data flow.

NOTE | *Before kicking off the stream, we will set a small optimization that will allow this to run better on a single machine. This simply limits the number of output partitions after a shuffle, a concept we discussed in the last chapter. We discuss this in Part VI of the book.*

```
spark.conf.set("spark.sql.shuffle.partitions", "5")
```

Streaming actions are a bit different from our conventional static action because we're going to be populating data somewhere instead of just calling something like count (which doesn't make any sense on a stream anyways). The action we will use will out to an in-memory table that we will update after each trigger. In this case, each trigger is based on an individual file (the read option that we set). Spark will mutate the data in the in-memory table such that we will always have the highest value as specified in our aggregation above.

```
%scala
purchaseByCustomerPerHour.writeStream
   .format("memory") // memory = store in-memory table
   .queryName("customer_purchases") // counts = name of the in-memory table
   .outputMode("complete") // complete = all the counts should be in the table
   .start()
```

```
%python
purchaseByCustomerPerHour.writeStream\
   .format("memory")\
   .queryName("customer_purchases")\
   .outputMode("complete")\
   .start()
```

Once we start the stream, we can run queries against the stream to debug what our result will look like if we were to write this out to a production sink.

```
%scala
spark.sql("""
   SELECT *
   FROM customer_purchases
   ORDER BY `sum(total_cost)` DESC
  """)
  .show(5)
```

```
%python
spark.sql("""
   SELECT *
   FROM customer_purchases
   ORDER BY `sum(total_cost)` DESC
  """)\
  .show(5)
```

You'll notice that as we read in more data - the composition of our table changes! With each file the results may or may not be changing based on the data. Naturally since we're grouping customers we hope to see an increase in the top customer purchase amounts over time (and do for a period of time!). Another option you can use is to just simply write the results out to the console.

```
purchaseByCustomerPerHour.writeStream
   .format("console")
   .queryName("customer_purchases_2")
   .outputMode("complete")
   .start()
```

Neither of these streaming methods should be used in production but they do make for convenient demonstration of Structured Streaming's power. Notice how this window is built on event time as well, not the time at which the data Spark processes the data. This was one of the shortcoming of Spark Streaming that Structured Streaming as resolved. We cover Structured Streaming in depth in Part V of this book.

Machine Learning and Advanced Analytics

Another popular aspect of Spark is its ability to perform large scale machine learning with a built-in library of machine learning algorithms called MLlib. MLlib allows for preprocessing, munging, training of models, and making predictions at scale on data. You can even use models trained in MLlib to make predictions in Strucutred Streaming. Spark provides a sophisticated machine learning API for performing a variety of machine learning tasks, from classification to regression, clustering to deep learning. To demonstrate this functionality, we will perform some basic clustering on our data using a common algorithm called K-Means.

WHAT IS K-MEANS? K-means is a clustering algorithm where "K" centers are randomly assigned within the data. The points closest to that point are then "assigned" to a particular cluster. Then a new center for this cluster is computed (called a centroid). We then label the points closest to that centroid, to the centroid's class, and shift the centroid to the new center of that cluster of points. We repeat this process for a finite set of iterations or until convergence (where our centroid and clusters stop changing.

Spark includes a number of preprocessing methods out of the box. To demonstrate these methods, we will start with some raw data, build up transformations before getting the data into the right format at which point we can actually train our model and then serve predictions.

```
staticDataFrame.printSchema()
```

```
root
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: integer (nullable = true)
 |-- InvoiceDate: timestamp (nullable = true)
 |-- UnitPrice: double (nullable = true)
 |-- CustomerID: double (nullable = true)
 |-- Country: string (nullable = true)
```

Machine learning algorithms in MLlib require data to be represented as numerical values. Our current data is represented by a variety of different types including timestamps, integers, and strings. Therefore we need to transform this data into some numerical representation. In this instance, we will use several DataFrame transformations to manipulate our date data.

```
%scala
import org.apache.spark.sql.functions.date_format
val preppedDataFrame = staticDataFrame
  .na.fill(0)
  .withColumn("day_of_week", date_format($"InvoiceDate", "EEEE"))
  .coalesce(5)
```

```
%python
from pyspark.sql.functions import date_format, col
preppedDataFrame = staticDataFrame\
  .na.fill(0)\
  .withColumn("day_of_week", date_format(col("InvoiceDate"), "EEEE"))\
  .coalesce(5)
```

Now we are also going to need to split our data into training and test sets. In this instance we are going to do this manually by the data that a certain purchase occurred however we could also leverage MLlib's transformation APIs to create a training and test set via train validation splits or cross validation. These topics are covered extensively in Part VI of this book.

```
%scala
val trainDataFrame = preppedDataFrame
  .where("InvoiceDate < '2011-07-01'")
val testDataFrame = preppedDataFrame
  .where("InvoiceDate >= '2011-07-01'")
```

```
%python
trainDataFrame = preppedDataFrame\
.where("InvoiceDate < '2011-07-01'")
   testDataFrame = preppedDataFrame\
   .where("InvoiceDate >= '2011-07-01'")
```

Now that we prepared our data, let's split it into a training and test set. Since this is a time-series set of data, we will split by an arbitrary date in the dataset. While this may not be the optimal split for our training and test, for the intents and purposes of this example it will work just fine. We'll see that this splits our dataset roughly in half.

```
trainDataFrame.count()
```

```
trainDataFrame.count()
```

Now these transformations are DataFrame transformations, covered extensively in part two of this book. Spark's MLlib also provides a number of transformations that allow us to automate some of our general transformations. One such transformer is a `StringIndexer`.

```
%scala
import org.apache.spark.ml.feature.StringIndexer
val indexer = new StringIndexer()
   .setInputCol("day_of_week")
   .setOutputCol("day_of_week_index")
```

```
%python
from pyspark.ml.feature import StringIndexer
indexer = StringIndexer()\
   .setInputCol("day_of_week")\
   .setOutputCol("day_of_week_index")
```

This will turn our days of weeks into corresponding numerical values. For example, Spark may represent Saturday as 6 and Monday as 1. However with this numbering scheme, we are implicitly stating that Saturday is greater than Monday (by pure numerical values). This is obviously incorrect. Therefore we need to use a `OneHotEncoder` to encode each of these values as their own column. These boolean flags state whether that day of week is the relevant day of the week.

```
%scala
import org.apache.spark.ml.feature.OneHotEncoder
val encoder = new OneHotEncoder()
   .setInputCol("day_of_week_index")
   .setOutputCol("day_of_week_encoded")
```

```
%python
from pyspark.ml.feature import OneHotEncoder
encoder = OneHotEncoder()\
   .setInputCol("day_of_week_index")\
   .setOutputCol("day_of_week_encoded")
```

Each of these will result in a set of columns that we will "assemble" into a vector. All machine learning algorithms in Spark take as input a `Vector` type, which must be a set of numerical values.

```
%scala
import org.apache.spark.ml.feature.VectorAssembler
val vectorAssembler = new VectorAssembler()
   .setInputCols(Array("UnitPrice", "Quantity", "day_of_week_encoded"))
   .setOutputCol("features")
```

```
%python
from pyspark.ml.feature import VectorAssembler
vectorAssembler = VectorAssembler()\
   .setInputCols(["UnitPrice", "Quantity", "day_of_week_encoded"])\
   .setOutputCol("features")
```

We can see that we have 3 key features, the price, the quantity, and the day of week. Now we'll set this up into a pipeline so any future data we need to transform can go through the exact same process.

```
%scala
import org.apache.spark.ml.Pipeline
val transformationPipeline = new Pipeline()
   .setStages(Array(indexer, encoder, vectorAssembler))
```

```
%python
from pyspark.ml import Pipeline
transformationPipeline = Pipeline()\
   .setStages([indexer, encoder, vectorAssembler])
```

Now preparing for training is a two step process. We first need to fit our transformers to this dataset. We cover this in depth, but basically our `StringIndexer` needs to know how many unique values there are to be index. Once those exist, encoding is easy but Spark must look at all the distinct values in the column to be indexed in order to store those values later on.

```
%scala
val fittedPipeline = transformationPipeline.fit(trainDataFrame)
```

```
%python
fittedPipeline = transformationPipeline.fit(trainDataFrame)
```

Once we fit the training data, we are now create to take that fitted pipeline and use it to transform all of our data in a consistent and repeatable way.

```
%scala
val transformedTraining = fittedPipeline.transform(trainDataFrame)
```

```
%python
transformedTraining = fittedPipeline.transform(trainDataFrame)
```

At this point, it's worth mentioning that we could have included our model training in our pipeline. We chose not to in order to demonstrate a use case for caching the data. At this point, we're going to perform some hyperparameter tuning on the model, since we do not want to repeat the exact same transformations over and over again, we'll leverage an optimization we discuss in Part IV of this book, caching.

This will put a copy of this intermediately transformed dataset into memory, allowing us to repeatedly access it at much lower cost than running the entire pipeline again. If you're curious to see how much of a difference this makes, skip this line and run the training without caching the data. Then try it after caching, you'll see the results are significant.

```
transformedTraining.cache()
```

Now we have a training set, now it's time to train the model. First we'll import the relevant model that we'd like to use and instantiate it.

```
%scala
import org.apache.spark.ml.clustering.KMeans
val kmeans = new KMeans()
  .setK(20)
  .setSeed(1L)
```

```
%python
from pyspark.ml.clustering import KMeans
kmeans = KMeans()\
  .setK(20)\
  .setSeed(1L)
```

In Spark, training machine learning models is a two phase process. First we initialize an untrained model, then we train it. There are always two types for every algorithm in MLlib's DataFrame API. They following the naming pattern of `Algorithm`, for the untrained version, and `AlgorithmModel` for the trained version. In our case, this is `KMeans` and then `KMeansModel`.

Predictors in MLlib's DataFrame API share roughly the same interface that we saw above with our preprocessing transformers like the `StringIndexer`. This should come as no surprise because it makes training an entire pipeline (which includes the model) simple. In our case we want to do things a bit more step by step, so we chose to not do this at this point.

```
%scala
val kmModel = kmeans.fit(transformedTraining)
```

```
%python
kmModel = kmeans.fit(transformedTraining)
```

We can see the resulting cost at this point. Which is quite high, that's likely because we didn't necessary scale our data or transform.

```
kmModel.computeCost(transformedTraining)
```

```
%scala
val transformedTest = fittedPipeline.transform(testDataFrame)
```

```
%python
transformedTest = fittedPipeline.transform(testDataFrame)
```

```
kmModel.computeCost(transformedTest)
```

Naturally we could continue to improve this model, layering more preprocessing as well as performing hyperparameter tuning to ensure that we're getting a good model. We leave that discussion for Part VI of this book.

Lower Level APIs

Spark includes a number of lower level primitives to allow for arbitrary Java and Python object manipulation via Resilient Distributed Datasets (RDDs). Virtually everything in Spark is built on top of RDDs. As we will cover in the next chapter, DataFrame operations are built on top of RDDs and compile down to these lower level tools for convenient and extremely efficient distributed execution. There are some things that you might use RDDs for, especially when you're reading or manipulating raw data, but for the most part you should stick to the Structured APIs. RDDs are lower level that DataFrames because they reveal physical execution characteristics (like partitions) to end users.

RDDs are available in Scala as well as Python. However, they're not equivalent. This differs from the DataFrame API (where the execution characteristics are the same) due to some underlying implementation details. We cover lower level APIs, including RDDs in Part IV of this book. As end users, you shouldn't need to use RDDs much in order to perform many tasks unless you're maintaining older Spark code. There are basically no instances in modern Spark where you should be using RDDs instead of the structured APIs beyond manipulating some very raw unprocessed and unstructured data.

One thing you might use RDDs for is to parallelize raw data you have stored in memory on the driver machine. For instance let's parallelize some simple numbers and create a DataFrame after we do so. We can then convert that to a DataFrame to use it with other DataFrames.

```
%scala
spark.sparkContext.parallelize(Seq(1, 2, 3)).toDF()
```

```
%python
from pyspark.sql import Row
spark.sparkContext.parallelize([Row(1), Row(2), Row(3)]).toDF()
```

SparkR

SparkR is a tool for running R on Spark. It follows the same principles as all of Spark's other language bindings. To use SparkR, we simply import it into our environment and run our code. It's all very similar to the Python API except that it follows R's syntax instead of Python. For the most part, almost everything available in Python is available in SparkR.

```
%r
library(SparkR)
sparkDF <- read.df("/mnt/defg/flight-data/csv/2015-summary.csv",
        source = "csv", header="true", inferSchema = "true")
take(sparkDF, 5)
```

```
%r
collect(orderBy(sparkDF, "count"), 20)
```

R users can also leverage other R libraries like the pipe operator in magrittr in order to make Spark transformations a bit more R like. This can make it easy to use with other libraries like ggplot for more sophisticated plotting.

```
%r
library(magrittr)
sparkDF %>%
   orderBy(desc(sparkDF$count)) %>%
   groupBy("ORIGIN_COUNTRY_NAME") %>%
   count() %>%
   limit(10) %>%
   collect()
```

We cover SparkR more in the Ecosystem Part of this book along with short discussion of PySpark specifics (PySpark is covered heavily through this book), and the new sparklyr package.

Spark's Ecosystem and Packages

One of the best parts about Spark is the ecosystem of packages and tools that the community has created. Some of these tools even move into the core Spark project as they mature and become widely used. The list of packages is rather large at over 300 at the time of this writing and more are added frequently. The largest index of Spark Packages can be found at spark-packages.org, where any user can publish to this package repository. There are also various other projects and packages that can be found through the web, for example on GitHub.

CHAPTER 3: Working with Different Types of Data

In the previous chapter, we covered basic DataFrame concepts and abstractions. This chapter will cover building expressions, which are the bread and butter of Spark's structured operations.

This chapter will cover working with a variety of different kinds of data including:

- Booleans
- Numbers
- Strings
- Dates and Timestamps
- Handling Null
- Complex Types
- User Defined Functions

Where to Look for APIs

Before we get started, it's worth explaining where you as a user should start looking for transformations. Spark is a growing project and any book (including this one) is a snapshot in time. Therefore it is our priority to educate you as a user as to where you should look for functions in order to transform your data. The key places to look for transformations are:

`DataFrame` (`Dataset`) Methods. This is actually a bit of a trick because a DataFrame is just a Dataset of `Row` types so you'll actually end up looking at the `Dataset` methods. These are available here.

`Dataset` sub-modules like `DataFrameStatFunctions` and `DataFrameNaFunctions` have more methods that solve specific sets of problems. For example, `DataFrameStatFunctions` holds a variety of statistically related functions while `DataFrameNaFunctions` refers to functions that are relevant when working with null data.

- Null Functions available here.
- Stat Functions vailable here.

`Column` Methods. These were introduced for the most part in the previous chapter are hold a variety of general column related methods like `alias` or `contains`. These are available here.

`org.apache.spark.sql.functions` contains a variety of functions for a variety of different data types. Often you'll see the entire package imported because they are used so often. These are available here.

Now this may feel a bit overwhelming but have no fear, the majority of these functions are ones that you will find in SQL and analytics systems. All of these tools exist to achieve one purpose, to transform rows of data in one format or structure to another. This may create more rows or reduce the number of rows available. To get started, let's read in the DataFrame that we'll be using for this analysis.

```
%scala
val df = spark.read.format("csv")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("/mnt/defg/retail-data/by-day/2010-12-01.csv")
df.printSchema()
df.createOrReplaceTempView("dfTable")
```

```
%python
df = spark.read.format("csv")\
  .option("header", "true")\
  .option("inferSchema", "true")\
  .load("/mnt/defg/retail-data/by-day/2010-12-01.csv")
df.printSchema()
df.createOrReplaceTempView("dfTable")
```

Here's the result of the schema and a small sample of the data.

```
root
   |-- InvoiceNo: string (nullable = true)
   |-- StockCode: string (nullable = true)
   |-- Description: string (nullable = true)
   |-- Quantity: integer (nullable = true)
   |-- InvoiceDate: timestamp (nullable = true)
   |-- UnitPrice: double (nullable = true)
   |-- CustomerID: double (nullable = true)
   |-- Country: string (nullable = true)
```

```
+---------+---------+--------------------+--------+-------------------+---------+----------+--------------+
|InvoiceNo|StockCode|         Description|Quantity|        InvoiceDate|UnitPrice|CustomerID|       Country|
+---------+---------+--------------------+--------+-------------------+---------+----------+--------------+
|   536365|   85123A|WHITE HANGING HEA...|       6|2010-12-01 08:26:00|     2.55|   17850.0|United Kingdom|
|   536365|    71053| WHITE METAL LANTERN|       6|2010-12-01 08:26:00|     3.39|   17850.0|United Kingdom|
|   536365|   84406B|CREAM CUPID HEART...|       8|2010-12-01 08:26:00|     2.75|   17850.0|United Kingdom|
|   536365|   84029G|KNITTED UNION FLA...|       6|2010-12-01 08:26:00|     3.39|   17850.0|United Kingdom|
...
|   536367|    21754|HOME BUILDING BLO...|       3|2010-12-01 08:34:00|     5.95|   13047.0|United Kingdom|
|   536367|    21755|LOVE BUILDING BLO...|       3|2010-12-01 08:34:00|     5.95|   13047.0|United Kingdom|
|   536367|    21777|RECIPE BOX WITH M...|       4|2010-12-01 08:34:00|     7.95|   13047.0|United Kingdom|
+---------+---------+--------------------+--------+-------------------+---------+----------+--------------+
```

Converting to Spark Types

One thing you'll see us do throughout this chapter is convert native types into Spark types. We do this with our first function, the `lit` function. The lit with take a type in a native language and convert it into the Spark representation. Here's how we can convert a couple of different kinds of Scala and Python values into their respective Spark types.

```
%scala
import org.apache.spark.sql.functions.lit
df.select(lit(5), lit("five"), lit(5.0))
```

```
%python
from pyspark.sql.functions import lit
df.select(lit(5), lit("five"), lit(5.0))
```

There's no equivalent function necessary in SQL, so we can just use the values directly.

```
%sql
SELECT 5, "five", 5.0
```

Working with Booleans

Booleans are foundational when it comes to data analysis because they are the foundation for all filtering. Boolean statements consist of four elements: and, or, true and false. We use these simple structures to build logical statements that evaluate to either true or false. These statements are often used as conditional requirements where a row of data must either pass this test (evaluate to true) or else it will be filtered out.

Let's use our retail dataset to explore working with booleans. We can specify equality as well as less or greater than.

```
%scala
import org.apache.spark.sql.functions.col
df.where(col("InvoiceNo").equalTo(536365))
  .select("InvoiceNo", "Description")
  .show(5, false)
```

NOTE | *Scala has some particular semantics around the use of `==` and `===`. In Spark, if you wish to filter by equality you should use `===` (equal) or `=!=` (not equal). You can also use `not` function and the `equalTo` method.*

```
%scala
import org.apache.spark.sql.functions.col
df.where(col("InvoiceNo") === 536365)
  .select("InvoiceNo", "Description")
  .show(5, false)
```

Python keeps a more conventional notation.

```
%python
from pyspark.sql.functions import col
   df.where(col("InvoiceNo") != 536365)\
   .select("InvoiceNo", "Description")\
   .show(5, False)
```

```
+---------+-----------------------------+
|InvoiceNo|Description                  |
+---------+-----------------------------+
|536366   |HAND WARMER UNION JACK       |
...
|536367   |POPPY'S PLAYHOUSE KITCHEN    |
+---------+-----------------------------+
```

Another option, and probably the cleanest, is to specify the predicate as an expression in a string. This is valid for Python or Scala. Note that this also gives us access to another way of expressing "does not equal".

```
df.where("InvoiceNo = 536365")
   .show(5, false)
df.where("InvoiceNo <> 536365")
   .show(5, false)
```

Now we mentioned that we can specify boolean expressions with multiple parts when we use `and` or `or`. In Spark you should always chain together `and` filters as a sequential filter.

The reason for this is that even if boolean expressions are expressed serially (one after the other) Spark will flatten all of these filters into one statement and perform the filter at the same time, creating the `and` statement for us. While you may specify your statements explicitly using `and` if you like, it's often easier to reason about and to read if you specify them serially. `or` statements need to be specified in the same statement.

```
%scala
val priceFilter = col("UnitPrice") > 600
val descripFilter = col("Description").contains("POSTAGE"

df.where(col("StockCode").isin("DOT"))
   .where(priceFilter.or(descripFilter))
   .show()
```

```
%python
from pyspark.sql.functions import instr
priceFilter = col("UnitPrice") > 600
descripFilter = instr(df.Description, "POSTAGE") >= 1
df.where(df.StockCode.isin("DOT"))\
   .where(priceFilter | descripFilter)\
   .show()
```

```
%sql
SELECT
   *
FROM dfTable
WHERE
   StockCode in (“DOT”) AND
   (UnitPrice > 600 OR
   instr(Description, “POSTAGE”) >= 1)
```

```
+---------+---------+--------------+--------+-------------------+---------+----------+--------------+
|InvoiceNo|StockCode|   Description|Quantity|        InvoiceDate|UnitPrice|CustomerID|       Country|
+---------+---------+--------------+--------+-------------------+---------+----------+--------------+
|   536544|      DOT|DOTCOM POSTAGE|       1|2010-12-01 14:32:00|   569.77|      null|United Kingdom|
|   536592|      DOT|DOTCOM POSTAGE|       1|2010-12-01 17:06:00|   607.49|      null|United Kingdom|
+---------+---------+--------------+--------+-------------------+---------+----------+--------------+
```

Boolean expressions are not just reserved to filters. In order to filter a `DataFrame` we can also just specify a boolean column.

```
%scala
val DOTCodeFilter = col("StockCode") === "DOT"
val priceFilter = col("UnitPrice") > 600
val descripFilter = col("Description").contains("POSTAGE")
df.withColumn("isExpensive",
   DOTCodeFilter.and(priceFilter.or(descripFilter)))
  .where("isExpensive")
  .select("unitPrice", "isExpensive")
  .show(5)
```

```
%python
from pyspark.sql.functions import instr
DOTCodeFilter = col("StockCode") == "DOT"
priceFilter = col("UnitPrice") > 600
descripFilter = instr(col("Description"), "POSTAGE") >= 1
df.withColumn("isExpensive",
DOTCodeFilter & (priceFilter | descripFilter))\
.where("isExpensive")\
.select("unitPrice", "isExpensive")\
.show(5)
```

```
%sql
SELECT
   UnitPrice,
   (StockCode = 'DOT' AND
   (UnitPrice > 600 OR
   instr(Description, "POSTAGE") >= 1)) as isExpensive
   FROM dfTable
WHERE
   (StockCode = 'DOT' AND
   (UnitPrice > 600 OR
   instr(Description, "POSTAGE") >= 1))
```

Notice how we did not have to specify our filter as an expression and how we could use a column name without any extra work.

If you're coming from a SQL background all of these statements should seem quite familiar. Indeed, all of them can be expressed as a `where` clause. In fact, it's often easier to just express filters as SQL statements than using the programmatic `DataFrame` interface and Spark SQL allows us to do this without paying any performance penalty. For example, the two following statements are equivalent.

```
%scala
import org.apache.spark.sql.functions.{expr, not, col}
df.withColumn("isExpensive", not(col("UnitPrice").leq(250)))
  .filter("isExpensive")
  .select("Description", "UnitPrice")
  .show(5)
df.withColumn("isExpensive", expr("NOT UnitPrice <= 250"))
  .filter("isExpensive")
  .select("Description", "UnitPrice")
  .show(5)
```

Here's our state definition.

```
%python
from pyspark.sql.functions import expr
df.withColumn("isExpensive", expr("NOT UnitPrice <= 250"))\
  .where("isExpensive")\
  .select("Description", "UnitPrice")
  .show(5)
```

WARNING | *One "gotcha" that can come up is working with null data when creating boolean expressions. If there is a null in your data, you're going to have to treat things a bit differently. Here's how we can ensure that we perform a null safe equivalence test.*

```
df.where(col("Description").eqNullSafe("hello")).show()
```

While not currently available (Spark 2.2), `IS [NOT] DISTINCT FROM` will be coming in Spark 2.3 to do the same thing in SQL.

Converting to Spark Types

When working with big data, the second most common task you will do after filtering things is counting things. For the most part, we simply need to express our computation and that should be valid assuming we're working with numerical data types.

To fabricate a contrived example, let's imagine that we found out that we misrecorded the quantity in our retail dataset and true quantity is equal to (the current quantity * the unit price) ^ 2 + 5. This will introduce our first numerical function as well the pow function that raises a column to the expressed power.

```
%scala
import org.apache.spark.sql.functions.{expr, pow}
val fabricatedQuantity = pow(col("Quantity") * col("UnitPrice"), 2) + 5
df.select(
   expr("CustomerId"),
   fabricatedQuantity.alias("realQuantity"))
  .show(2)
```

```
%python
from pyspark.sql.functions import expr, pow
fabricatedQuantity = pow(col("Quantity") * col("UnitPrice"), 2) + 5
df.select(
   expr("CustomerId"),
   fabricatedQuantity.alias("realQuantity"))\
.show(2)
```

```
+----------+------------------+
|CustomerId|      realQuantity|
+----------+------------------+
|   17850.0|239.08999999999997|
|   17850.0|          418.7156|
+----------+------------------+
```

You'll notice that we were able to multiply our columns together because they were both numerical. Naturally we can add and subtract as necessary as well. In fact we can do all of this a SQL expression as well.

```
%scala
df.selectExpr(
   "CustomerId",
   "(POWER((Quantity * UnitPrice), 2.0) + 5) as realQuantity")
   .show(2)
```

```
%python
df.selectExpr(
   "CustomerId",
   "(POWER((Quantity * UnitPrice), 2.0) + 5) as realQuantity" )
   .show(2)
```

```
%sql
SELECT
   customerId,
   (POWER((Quantity * UnitPrice), 2.0) + 5) as realQuantity
FROM dfTable
```

Another common numerical task is rounding. Now if you'd like to just round to a whole number, often times you can cast it to an integer and that will work just fine. However Spark also has more detailed functions for performing this explicitly and to a certain level of precision. In this case we will round to one decimal place.

```
%scala
import org.apache.spark.sql.functions.{round, bround}
df.select(
   round(col("UnitPrice"), 1).alias("rounded"),
   col("UnitPrice"))
  .show(5)
```

By default, the `round` function will round up if you're exactly in between two numbers. You can round down with the `bround`.

```
%scala
import org.apache.spark.sql.functions.lit
df.select(
   round(lit("2.5")),
   bround(lit("2.5")))
  .show(2)
```

```
%python
from pyspark.sql.functions import lit, round, bround
df.select(
   round(lit("2.5")),
   bround(lit("2.5")))\
  .show(2)
```

```
%sql
SELECT
   round(2.5),
   bround(2.5)
```

```
+-------------+--------------+
|round(2.5, 0)|bround(2.5, 0)|
+-------------+--------------+
|          3.0|           2.0|
|          3.0|           2.0|
+-------------+--------------+
```

Another numerical task is to compute the correlation of two columns. For example, we can see the Pearson Correlation Coefficient for two columns to see if cheaper things are typically bought in greater quantities. We can do this through a function as well as through the DataFrame statistic methods.

```
%scala
import org.apache.spark.sql.functions.{corr}
df.stat.corr("Quantity", "UnitPrice")
df.select(corr("Quantity", "UnitPrice")).show()
```

```
%python
from pyspark.sql.functions import corr
df.stat.corr("Quantity", "UnitPrice")
df.select(corr("Quantity", "UnitPrice")).show()
```

```
%sql
SELECT
   corr(Quantity, UnitPrice)
FROM
   dfTable
```

```
+-------------------------+
|corr(Quantity, UnitPrice)|
+-------------------------+
|     -0.04112314436835551|
+-------------------------+
```

A common task is to compute summary statistics for a column or set of columns. We can use the `describe` method to achieve exactly this. This will take all numeric columns and calculate the count, mean, standard deviation, min, and max. This should be used primarily for viewing in the console as the schema may change in the future.

```
%scala
df.describe().show()
```

```
%python
df.describe().show()
```

```
+-------+------------------+------------------+------------------+
|Summary|          Quantity|         UnitPrice|        CustomerID|
+-------+------------------+------------------+------------------+
|  count|              3108|              3108|              1968|
|   mean| 8.627413127413128| 4.151946589446603|15661.388719512195|
| stddev|26.371821677029203|15.638659854603892|1854.4496996893627|
|    min|               -24|               0.0|           12431.0|
|    max|               600|            607.49|           18229.0|
+-------+------------------+------------------+------------------+
```

If you need these exact numbers you can also perform this as an aggregation yourself by importing the functions and applying them to the columns that you need.

```
%scala
import org.apache.spark.sql.functions.{count, mean, stddev_pop, min, max}
```

```
%python
from pyspark.sql.functions import count, mean, stddev_pop, min, max
```

There are a number of statistical functions available in the StatFunctions Package. These are DataFrame methods that allow you to calculate a variety of different things. For instance, we can calculate either exact or approximate quantiles of our data using the `approxQuantile` method.

```
%scala
val colName = "UnitPrice"
val quantileProbs = Array(0.5)
val relError = 0.05
df.stat.approxQuantile("UnitPrice", quantileProbs, relError) // 2.51
```

```
%python
colName = "UnitPrice"
quantileProbs = [0.5]
relError = 0.05
df.stat.approxQuantile("UnitPrice", quantileProbs, relError) # 2.51
```

We can also use this to see a cross tabulation or frequent item pairs (Be careful, this output will be large and is omitted for this reason).

```
%scala
df.stat.crosstab("StockCode", "Quantity").show()
```

```
%python
df.stat.crosstab("StockCode", "Quantity").show()
```

```
%scala
df.stat.freqItems(Seq("StockCode", "Quantity")).show()
```

```
%python
df.stat.freqItems(["StockCode", "Quantity"]).show()
```

As a last note, we can also add a unique id to each row using the `monotonically_increasing_id` function. This function will generate a unique value for each row, starting with 0.

```
%scala
import org.apache.spark.sql.functions.monotonically_increasing_id
df.select(monotonically_increasing_id()).show(2)
```

```
%python
from pyspark.sql.functions import monotonically_increasing_id
df.select(monotonically_increasing_id()).show(2)
```

There are functions added every release and so by the time you're reading this, it may already not include everything. For instance, there are some random data generation tools (`rand() randn()`) that allow you to randomly generate data however there are potential determinism issues when doing so. Discussions of these challenges can be found on the Spark mailing list. There are also a number of more advanced tasks like bloom filtering and sketching algorithms available in the stat functions that we mentioned (and linked to) at the beginning of this chapter. Be sure to search the API documentation for more information and functions.

Working with Strings

String manipulation shows up in nearly every data flow and its worth explaining what you can do with strings. You may be manipulating log files performing regular expression extraction or substitution, or checking for simple string existence, or simply making all strings upper or lower case.

We will start with the last task as it's one of the simplest. The `initcap` function will capitalize every word in a given string when that word is separated from another via whitespace.

```
%scala
import org.apache.spark.sql.functions.{initcap}
df.select(initcap(col("Description"))).show(2, false)
```

```
%python
from pyspark.sql.functions import initcap
df.select(initcap(col("Description"))).show()
```

```
%sql
SELECT
   initcap(Description)
FROM
   dfTable
```

```
+---------------------------------+
|initcap(Description)             |
+---------------------------------+
|White Hanging Heart T-light Holder|
|White Metal Lantern              |
+---------------------------------+
```

As mentioned above, we can also quite simply lower case and upper case strings as well.

```
%scala
import org.apache.spark.sql.functions.{lower, upper}
df.select(
   col(“Description”),
   lower(col(“Description”)),
   upper(lower(col(“Description”))))
  .show(2)
```

```
%python
from pyspark.sql.functions import lower, upper
df.select(
   col(“Description”),
   lower(col(“Description”)),
   upper(lower(col(“Description”))))\
  .show(2)
```

```
%sql
SELECT
   Description,
   lower(Description),
   Upper(lower(Description))
FROM
   dfTable
```

```
+--------------------+--------------------+-------------------------+
|         Description|  lower(Description)|upper(lower(Description))|
+--------------------+--------------------+-------------------------+
|WHITE HANGING HEA...|white hanging hea...|     WHITE HANGING HEA...|
| WHITE METAL LANTERN| white metal lantern|      WHITE METAL LANTERN|
+--------------------+--------------------+-------------------------+
```

Another trivial task is adding or removing whitespace around a string. We can do this with `lpad`, `ltrim`, `rpad` and `rtrim`, `trim`.

```
%scala
import org.apache.spark.sql.functions.{lit, ltrim, rtrim, rpad, lpad,
trim}
df.select(
   ltrim(lit(" HELLO ")).as("ltrim"),
   rtrim(lit(" HELLO ")).as("rtrim"),
   trim(lit(" HELLO ")).as("trim"),
   lpad(lit("HELLO"), 3, " ").as("lp"),
   rpad(lit("HELLO"), 10, " ").as("rp"))
.show(2)
```

```
%python
from pyspark.sql.functions import lit, ltrim, rtrim, rpad, lpad, trim
df.select(
   ltrim(lit(" HELLO ")).alias("ltrim"),
   rtrim(lit(" HELLO ")).alias("rtrim"),
   trim(lit(" HELLO ")).alias("trim"),
   lpad(lit("HELLO"), 3, " ").alias("lp"),
   rpad(lit("HELLO"), 10, " ").alias("rp"))\
  .show(2)
```

```
%sql
SELECT
   ltrim(' HELLLOOOO '),
   rtrim(' HELLLOOOO '),
   trim(' HELLLOOOO '),
   lpad('HELLOOOO ', 3, ' '),
   rpad('HELLOOOO ', 10, ' ')
FROM
   dfTable
```

```
+---------+---------+-----+---+----------+
|    ltrim|    rtrim| trim| lp|        rp|
+---------+---------+-----+---+----------+
|HELLO    |    HELLO|HELLO| HE|HELLO     |
|HELLO    |    HELLO|HELLO| HE|HELLO     |
+---------+---------+-----+---+----------+
```

You'll notice that if `lpad` or `rpad` takes a number less than the length of the string, it will always remove values from the right side of the string.

Regular Expressions

Probably one of the most frequently performed tasks is searching for the existence of one string on another or replacing all mentions of a string with another value. This is often done with a tool called "Regular Expressions" that exist in many programming languages. Regular expressions give the user an ability to specify a set of rules to use to either extract values from a string or replace them with some other values.

Spark leverages the complete power of Java Regular Expressions. The Java RegEx syntax departs slightly from other programming languages so it is worth reviewing before putting anything into production. There are two key functions in Spark that you'll need to perform regular expression tasks: `regexp_extract` and `regexp_replace`. These functions extract values and replace values respectively.

Let's explore how to use the `regexp_replace` function to replace substitute colors names in our description column.

```
%scala
import org.apache.spark.sql.functions.regexp_replace
val simpleColors = Seq("black", "white", "red", "green", "blue")
val regexString = simpleColors.map(_.toUpperCase).mkString("|")
// the | signifies `OR` in regular expression syntax
df.select(
   regexp_replace(col("Description"), regexString, "COLOR")
   .alias("color_cleaned"),
   col("Description"))
.show(2)
```

```
%python
from pyspark.sql.functions import regexp_replace
regex_string = "BLACK|WHITE|RED|GREEN|BLUE"
df.select(
   regexp_replace(col("Description"), regex_string, "COLOR")
   .alias("color_cleaned"),
   col("Description"))\
.show(2)
```

```
%sql
SELECT
   regexp_replace(Description, 'BLACK|WHITE|RED|GREEN|BLUE', 'COLOR')
   as color_cleaned,
   Description
FROM
   dfTable
```

```
+--------------------+--------------------+
|       color_cleaned|         Description|
+--------------------+--------------------+
|COLOR HANGING HEA...|WHITE HANGING HEA...|
| COLOR METAL LANTERN| WHITE METAL LANTERN|
+--------------------+--------------------+
```

Another task may be to replace given characters with other characters. Building this as regular expression could be tedious so Spark also provides the translate function to replace these values. This is done at the character level and will replace all instances of a character with the indexed character in the replacement string.

```
%scala
import org.apache.spark.sql.functions.translate
df.select(
   translate(col("Description"), "LEET", "1337"),
   col("Description"))
   .show(2)
```

```
%python
from pyspark.sql.functions import translate
df.select(
   translate(col("Description"), "LEET", "1337"),
   col("Description"))\
.show(2)
```

```
%sql
SELECT
   translate(Description, 'LEET', '1337'),
   Description
FROM
   dfTable
```

```
+-----------------------------------+--------------------+
|translate(Description, LEET, 1337)|         Description|
+-----------------------------------+--------------------+
|               WHI73 HANGING H3A...|WHITE HANGING HEA...|
|                WHI73 M37A1 1AN73RN| WHITE METAL LANTERN|
+-----------------------------------+--------------------+
```

We can also perform something similar like pulling out the first mentioned color.

```
%scala
import org.apache.spark.sql.functions.regexp_extract
val regexString = simpleColors
   .map(_.toUpperCase)

   .mkString("(", "|", ")")
// the | signifies OR in regular expression syntax
df.select(
   regexp_extract(col("Description"), regexString, 1)
   .alias("color_cleaned"),
   col("Description"))
.show(2)
```

```
%python
from pyspark.sql.functions import regexp_extract
extract_str = "(BLACK|WHITE|RED|GREEN|BLUE)"
   df.select(
   regexp_extract(col("Description"), extract_str, 1)
   .alias("color_cleaned"),
   col("Description"))\
.show(2)
```

```
%sql
SELECT
   regexp_extract(Description, '(BLACK|WHITE|RED|GREEN|BLUE)', 1),
   Description
FROM
   dfTable
```

```
+-------------+--------------------+
|color_cleaned|         Description|
+-------------+--------------------+
|        WHITE|WHITE HANGING HEA...|
|        WHITE| WHITE METAL LANTERN|
+-------------+--------------------+
```

Sometimes, rather than extracting values, we simply want to check for existence. We can do this with the `contains` method on each column. This will return a boolean declaring whether it can find that string in the column's string.

```
%scala
val containsBlack = col("Description").contains("BLACK")
val containsWhite = col("DESCRIPTION").contains("WHITE")
df.withColumn("hasSimpleColor", containsBlack.or(containsWhite))
  .filter("hasSimpleColor")
  .select("Description")
  .show(3, false)
```

In Python we can use the `instr` function.

```
%python
from pyspark.sql.functions import instr
containsBlack = instr(col("Description"), "BLACK") >= 1
containsWhite = instr(col("Description"), "WHITE") >= 1
df.withColumn("hasSimpleColor", containsBlack | containsWhite)\
  .filter("hasSimpleColor")\
  .select("Description")\
  .show(3, False)
```

```
%sql
SELECT
   Description
FROM
   dfTable
WHERE
   instr(Description, 'BLACK') >= 1 OR
   instr(Description, 'WHITE') >= 1
```

```
+---------------------------------+
|Description                      |
+---------------------------------+
|WHITE HANGING HEART T-LIGHT HOLDER|
|WHITE METAL LANTERN              |
|RED WOOLLY HOTTIE WHITE HEART.   |
+---------------------------------+
only showing top 3 rows
```

This is trivial with just two values but gets much more complicated with more values.

Let's work through this in a more dynamic way and take advantage of Spark's ability to accept a dynamic number of arguments. When we convert a list of values into a set of arguments and pass them into a function, we use a language feature called varargs. This feature allows us to effectively unravel an array of arbitrary length and pass it as arguments to a function. This, coupled with `select` allows us to create arbitrary numbers of columns dynamically.

```
%scala
val simpleColors = Seq("black", "white", "red", "green", "blue")
val selectedColumns = simpleColors.map(color => {
   col("Description")
   .contains(color.toUpperCase)
   .alias(s"is_$color")
}):+expr("*") // could also append this value
df
   .select(selectedColumns:_*)
   .where(col("is_white").or(col("is_red")))
   .select("Description")
   .show(3, false)
```

```
   +---------------------------------+
   |Description                      |
   +---------------------------------+
   |WHITE HANGING HEART T-LIGHT HOLDER|
   |WHITE METAL LANTERN              |
   |RED WOOLLY HOTTIE WHITE HEART.   |
   +---------------------------------+
```

We can also do this quite easily in Python. In this case we're going to use a different function `locate` that returns the integer location (1 based location). We then convert that to a boolean before using it as a the same basic feature.

```
%python
from pyspark.sql.functions import expr, locate
simpleColors = ["black", "white", "red", "green", "blue"]
def color_locator(column, color_string):
   """This function creates a column declaring whether or not a given
    pySpark column contains the UPPERCASED color. Returns a new column
    type that can be used in a select statement."""
   return locate(color_string.upper(), column)\
      .cast("boolean")\
      .alias("is_" + c)
selectedColumns = [color_locator(df.Description, c) for c in simple-
Colors]
selectedColumns.append(expr("*")) # has to a be Column type
df\
.select(*selectedColumns)\
.where(expr("is_white OR is_red"))\
.select("Description")\
.show(3, False)
```

This simple feature is often one that can help you programmatically generate columns or boolean filters in a way that is simple to reason about and extend. We could extend this to calculating the smallest common denominator for a given input value or whether or not a number is a prime.

Working with Dates and Timestamps

Dates and times are a constant challenge in programming languages and databases. It's always necessary to keep track of timezones and make sure that formats are correct and valid. Spark does its best to keep things simple by focusing explicitly on two kinds of time related information. There are dates, which focus exclusively on calendar dates, and timestamps that include both date and time information. Spark, as we saw with our current dataset, will make a best effort to correctly identify column types, including dates and timestamps when we enable `inferSchema`. We can see that this worked quite well with our current dataset because it was able to identify and read our date format without us having to provide some specification for it.

```
df.printSchema()
```

```
root
|-- InvoiceNo: string (nullable = true)
|-- StockCode: string (nullable = true)
|-- Description: string (nullable = true)
|-- Quantity: integer (nullable = true)
|-- InvoiceDate: timestamp (nullable = true)
|-- UnitPrice: double (nullable = true)
|-- CustomerID: double (nullable = true)
|-- Country: string (nullable = true)
```

Now as we hinted at above, working with dates and timestamps closely relates to working with strings because we often store our timestamps or dates as strings and convert them into date types at runtime. This is less common when working with databases and structured data but much more common when we are working with text and csv files. We will experiment with that shortly.

WARNING | *There are a lot of caveats, unfortunately, when working with dates and timestamps, especially when it comes to timezone handling. In 2.1 and before, Spark will parse according to the machine's timezone if timezones are not explicitly specified in the value that you are parsing. You can set a session local timezone if necessary by setting `spark.conf.sessionLocalTimeZone` in the SQL configurations. This should be set according to the Java TimeZone format.*

While Spark will do this on a best effort basis, sometimes there will be no getting around working with strangely formatted dates and times. Now the key to reasoning about the transformations that you are going to need to apply is to ensure that you know exactly what type and format you have at each given step of the way. Another common gotcha is that Spark's TimestampType only supports second-level precision, this means that if you're going to be working with milliseconds or microseconds, you're going to have to work around this problem by potentially operating on them as longs. Any more precision when coercing to a TimestampType will be removed.

Spark can be a bit particular about what format you have at any given point in time. It's important to be explicit when parsing or converting to make sure there are no issues in doing so. At the end of the day, Spark is working with Java dates and timestamps and therefore conforms to those standards. Let's start with the basics and get the current date and the current timestamps.

```
%scala
import org.apache.spark.sql.functions.{current_date, current_timestamp}
val dateDF = spark.range(10)
   .withColumn("today", current_date())
   .withColumn("now", current_timestamp())
   dateDF.createOrReplaceTempView("dateTable")
```

```
%python
from pyspark.sql.functions import current_date, current_timestamp
dateDF = spark.range(10)\
.withColumn("today", current_date())\
.withColumn("now", current_timestamp())
dateDF.createOrReplaceTempView("dateTable")
```

```
dateDF.printSchema()
```

```
root
   |-- id: long (nullable = false)
   |-- today: date (nullable = false)
   |-- now: timestamp (nullable = false)
```

Now that we have a simple DataFrame to work with, let's add and subtract 5 days from today. These functions take a column and then the number of days to either add or subtract as the arguments.

```
%scala
import org.apache.spark.sql.functions.{date_add, date_sub}
dateDF
   .select(
    date_sub(col("today"), 5),
    date_add(col("today"), 5))
   .show(1)
```

```
%python
from pyspark.sql.functions import date_add, date_sub
dateDF\
   .select(
       date_sub(col("today"), 5),
       date_add(col("today"), 5))\
   .show(1)
```

```
%sql
SELECT
   date_sub(today, 5),
   date_add(today, 5)
FROM
   dateTable
```

```
+------------------+------------------+
|date_sub(today, 5)|date_add(today, 5)|
+------------------+------------------+
|        2017-06-12|        2017-06-22|
+------------------+------------------+
```

Another common task is to take a look at the difference between two dates. We can do this with the `datediff` function that will return the number of days in between two dates. Most often we just care about the days although since months can have a strange number of days there also exists a function `months_between` that gives you the number of months between two dates.

```
%scala
import org.apache.spark.sql.functions.{datediff, months_between, to_date}
dateDF
  .withColumn("week_ago", date_sub(col("today"), 7))
  .select(datediff(col("week_ago"), col("today")))
  .show(1)
dateDF
  .select(
   to_date(lit("2016-01-01")).alias("start"),
   to_date(lit("2017-05-22")).alias("end"))
  .select(months_between(col("start"), col("end")))
  .show(1)
```

```
%python
from pyspark.sql.functions import datediff, months_between, to_date
dateDF\
  .withColumn("week_ago", date_sub(col("today"), 7))\
  .select(datediff(col("week_ago"), col("today")))\
  .show(1)
dateDF\
  .select(
   to_date(lit("2016-01-01")).alias("start"),
   to_date(lit("2017-05-22")).alias("end"))\
  .select(months_between(col("start"), col("end")))\
  .show(1)
```

```
%sql
SELECT
   to_date('2016-01-01'),
   months_between('2016-01-01', '2017-01-01'),
   datediff('2016-01-01', '2017-01-01')
FROM
   dateTable
```

```
+--------------------------+
|datediff(week_ago, today)|
+--------------------------+
|                        -7|
+--------------------------+
+--------------------------+
|months_between(start,end)|
+--------------------------+
|              -16.67741935|
+--------------------------+
```

```
%scala
import org.apache.spark.sql.functions.{to_date, lit}
spark.range(5).withColumn("date", lit("2017-01-01"))
   .select(to_date(col("date")))
   .show(1)
```

```
%python
from pyspark.sql.functions import to_date, lit
spark.range(5).withColumn("date", lit("2017-01-01"))\
   .select(to_date(col("date")))\
   .show(1)
```

You'll notice that I introduced a new function above, the `to_date` function. The `to_date` function allows you to convert a string to a date, optionally with a specified format. We specify our format in the Java simpleDateFormat which will be important to reference if you use this function.

WARNING | *Spark will not throw an error if it cannot parse the date, it'll just return null. This can be a bit tricky in larger pipelines because you may be expecting your data in one format and getting it in another. To illustrate, let's take a look at the date format that has switched from year-month-day to year-day-month. Spark will fail to parse this date and silently return null instead.*

```
dateDF.select(to_date(lit("2016-20-12")),to_date(lit("2017-12-11"))).
show(1)
```

```
+------------------+------------------+
|to_date(2016-20-12)|to_date(2017-12-11)|
+------------------+------------------+
|              null|        2017-12-11|
+------------------+------------------+
```

We find this to be an especially tricky situation for bugs because some dates may match the correct format while others do not. See how above, the second date is show to be December 11th instead of the correct day, November 12th? Spark doesn't throw an error because it cannot know whether the days are mixed up or if that specific row is incorrect.

Let's fix this pipeline, step by step and come up with a robust way to avoid these issues entirely. The first step is to remember that we need to specify our date format according to the Java `SimpleDateFormat` standard as documented here.

We will use two functions to fix this, `to_date` and `to_timestamp`. The former optionally expects a format while the latter requires one.

```
import org.apache.spark.sql.functions.{unix_timestamp, from_unixtime}
val dateFormat = "yyyy-dd-MM"
val cleanDateDF = spark.range(1)
.select(
    to_date(lit("2017-12-11"), dateFormat)
       .alias("date"),
    to_date(lit("2017-20-12"), dateFormat)
       .alias("date2"))
cleanDateDF.createOrReplaceTempView("dateTable2")
```

```
%python
from pyspark.sql.functions import unix_timestamp, from_unixtime
dateFormat = "yyyy-dd-MM"
cleanDateDF = spark.range(1)\
   .select(
    to_date(unix_timestamp(lit("2017-12-11"), dateFormat).cast("time-
    stamp"))\
       .alias("date"),
    to_date(unix_timestamp(lit("2017-20-12"), dateFormat).cast("time-
    stamp"))\
       .alias("date2"))
cleanDateDF.createOrReplaceTempView("dateTable2")
```

```
+----------+----------+
|      date|     date2|
+----------+----------+
|2017-11-12|2017-12-20|
+----------+----------+
```

```
%sql
SELECT
   to_date(date, 'yyyy-dd-MM'),
     to_date(date2, 'yyyy-dd-MM'),
     to_date(date)
FROM
   dateTable2
```

Now let's use an example of `to_timestamp` which always requires a format to be specified.

```
%scala
import org.apache.spark.sql.functions.to_timestamp
cleanDateDF
   .select(
       to_timestamp(col("date"), dateFormat))
   .show()
```

```
%python
from pyspark.sql.functions import to_timestamp
cleanDateDF\
   .select(
       to_timestamp(col("date"), dateFormat))\
   .show()
```

```
+----------------------------------+
|to_timestamp(`date`, 'yyyy-dd-MM')|
+----------------------------------+
|               2017-11-12 00:00:00|
+----------------------------------+
```

We can check all of this from SQL.

```
%sql
SELECT
   to_timestamp(date, 'yyyy-dd-MM'),
   to_timestamp(date2, 'yyyy-dd-MM')
FROM
   dateTable2
```

Casting between dates and timestamps is simple in all languages, in SQL we would do it in the following way.

```
%sql
SELECT cast(to_date("2017-01-01", "yyyy-dd-MM") as timestamp)
```

Once we've gotten our date or timestamp into the correct format and type, comparing between them is actually quite easy. We just need to be sure to either use a date/timestamp type or specify our string according to the right format of `yyyy-MM-dd` if we're comparing a date.

```
cleanDateDF.filter(col("date2") > lit("2017-12-12")).show()
```

One minor point is that we can also set this as a string which Spark parses to a literal.

```
cleanDateDF.filter(col("date2") > "'2017-12-12'").show()
```

WARNING | *Implicit type casting is an easy way to shoot yourself in the foot, especially when dealing with null values or dates in different timezones or formats. We recommend that you parse them explicitly instead of relying on implicit changes.*

Working with Nulls in Data

As a best practice, you should always use nulls to represent missing or empty data in your DataFrames. Spark can optimize working with null values more than it can if you use empty strings or other values. The primary way of interacting with null values, at DataFrame scale, is to use the `.na` subpackage on a DataFrame. There are also several functions for performing operations and explicitly specifying how Spark should handle null values. See the previous chapter where we discuss ordering and the section on boolean expressions previously in this chapter.

WARNING | *Nulls are a challenge part of all programming and Spark is no exception. We recommend being explicit is always better than being implicit when handling null values. For instance, in this part of the book we saw how we can define columns as having null types. However, this comes with a catch. When we declare a column as not having a null time, that is not actually enforced. To reiterate, when you define a schema where all columns are declared to not have null values - Spark will not enforce that and will happily let null values into that column. The nullable signal is somply to help Spark SQL optimize for handling that column. If you have null values in columns that should not have null values, you can get an incorrect result or see strange exceptions that can be hard to debug.*

There are two things you can do with null values. You can explicitly drop nulls or you can fill them with a value (globally or on a per column basis). Let's experiment with each of these now.

Coalesce

Spark includes a function to allow you to select the first null value from a set of columns by using the coalesce function. In this case there are no null values, so it simply returns the first column.

```
%scala
import org.apache.spark.sql.functions.coalesce
df.select(coalesce(col("Description"), col("CustomerId"))).show()
```

```
%python
from pyspark.sql.functions import coalesce
df.select(coalesce(col("Description"), col("CustomerId"))).show()
```

Nullif, Ifnull, nvl, and nvl2

There are several SQL functions that allow us to achieve similar things. `ifnull` allows you to select the second value if the first is null, and defaults to the first. `nullif` allows you to return null if the two values are equal or else return the second if they are not. `nvl` will return the second value if the first is null, but defaults to the first. Lastly, `nvl2` will return the second value is the first is not null, otherwise it will return last specified value (`else_value` below).

```
%sql
SELECT
   ifnull(null, 'return_value'),
   nullif('value', 'value'),
   nvl(null, 'return_value'),
   nvl2('not_null', 'return_value', "else_value")
FROM dfTable
LIMIT 1
```

```
+------------+----+------------+------------+
|           a|   b|           c|           d|
+------------+----+------------+------------+
|return_value|null|return_value|return_value|
+------------+----+------------+------------+
```

Naturally, we can use these in select expressions on DataFrames as well.

Drop

The simplest is probably drop, which simply removes rows that contain nulls. The default is to drop any row where any value is null.

```
df.na.drop()
df.na.drop("any")
```

In SQL we have to do this column by column.

```
%sql
SELECT
   *
FROM
   dfTable
WHERE
   Description IS NOT NULL
```

Passing in "any" as an argument will drop a row if any of the values are null. Passing in "all" will only drop the row if all values are null or NaN for that row.

```
df.na.drop("all")
```

We can also apply this to certain sets of columns by passing in an array of columns.

```
%scala
df.na.drop("all", Seq("StockCode", "InvoiceNo"))
```

```
%python
df.na.drop("all", subset=["StockCode", "InvoiceNo"])
```

Fill

Fill allows you to fill one or more columns with a set of values. This can be done by specifying a map, specific value and a set of columns.

For example to fill all null values in String columns I might specify.

```
df.na.fill("All Null values become this string")
```

We could do the same for integer columns with `df.na.fill(5:Integer)` or for Doubles `df.na.fill(5:Double)`. In order to specify columns, we just pass in an array of column names like we did above.

```
%scala
df.na.fill(5, Seq("StockCode", "InvoiceNo"))
```

```
%python
df.na.fill("all", subset=["StockCode", "InvoiceNo"])
```

We can also do with with a Scala `Map` where the key is the column name and the value is the value we would like to use to fill null values.

```
%scala
val fillColValues = Map(
   "StockCode" -> 5,
   "Description" -> "No Value"
)
df.na.fill(fillColValues)
```

```
%python
fill_cols_vals = {
   "StockCode": 5,
   "Description" : "No Value"
}
df.na.fill(fill_cols_vals)
```

Replace

In addition to replacing null values like we did with `drop` and `fill`, there are more flexible options that we can use with more than just null values. Probably the most common use case is to replace all values in a certain column according to their current value. The only requirement is that this value be the same type as the original value.

```
%scala
df.na.replace("Description", Map("" -> "UNKNOWN"))
```

```
%python
df.na.replace([""], ["UNKNOWN"], "Description")
```

Ordering

As discussed in the previous chapter, you can use `asc_nulls_first`, `desc_nulls_first`, `asc_nulls_last`, or `desc_nulls_last` to specify where we would like our null values to appear in an ordered DataFrame.

Working with Complex Types

Complex types can help you organize and structure your data in ways that make more sense for the problem you are hoping to solve. There are three kinds of complex types, structs, arrays, and maps.

Structs

You can think of structs as DataFrames within DataFrames. A worked example will illustrate this more clearly. We can create a struct by wrapping a set of columns in parenthesis in a query.

```
df.selectExpr("(Description, InvoiceNo) as complex", "*")
```

```
df.selectExpr("struct(Description, InvoiceNo) as complex", "*")
```

```
%scala
import org.apache.spark.sql.functions.struct
val complexDF = df
.select(struct("Description", "InvoiceNo").alias("complex"))
complexDF.createOrReplaceTempView("complexDF")
```

```
%python
from pyspark.sql.functions import struct
complexDF = df\
.select(struct("Description", "InvoiceNo").alias("complex"))
complexDF.createOrReplaceTempView("complexDF")
```

We now have a DataFrame with a column `complex`. We can query it just as we might another DataFrame, the only difference is that we use a dot syntax to do so or the column method `getField`.

```
complexDF.select("complex.Description")
complexDF.select(col("complex").getField("Description")
```

We can also query all values in the struct with `*`. This brings up all the columns to the top level DataFrame.

```
complexDF.select("complex.*")
```

```
%sql
SELECT
   complex.*
FROM
   complexDF
```

Arrays

To define arrays, let's work through a use case. With our current data, our object is to take every single word in our `Description` column and convert that into a row in our DataFrame.

The first task is to turn our Description column into a complex type, an array.

split

We do this with the split function and specify the delimiter.

```
%scala
import org.apache.spark.sql.functions.split
df.select(split(col("Description"), " ")).show(2)
```

```
%python
from pyspark.sql.functions import split
df.select(split(col("Description"), " ")).show(2)
```

```
%sql
SELECT
   split(Description, ' ')
FROM
   dfTable
```

```
+---------------------+
|split(Description,  )|
+---------------------+
| [WHITE, HANGING, ...|
| [WHITE, METAL, LA...|
+---------------------+
```

This is quite powerful because Spark will allow us to manipulate this complex type as another column. We can also query the values of the array with a python-like syntax.

```
%scala
df.select(split(col("Description"), " ").alias("array_col"))
.selectExpr("array_col[0]")
.show(2)
```

```
%python
df.select(split(col("Description"), " ").alias("array_col"))\
.selectExpr("array_col[0]")\
.show(2)
```

```
%sql
SELECT
   split(Description, ' ')[0]
FROM
   dfTable
```

```
+------------+
|array_col[0]|
+------------+
|       WHITE|
|       WHITE|
+------------+
```

Array Length

We can query the array's length by querying for its size.

```
%scala
import org.apache.spark.sql.functions.size
df.select(size(split(col("Description"), " "))).show(2) // shows 5 and 3
```

```
%python
from pyspark.sql.functions import size
df.select(size(split(col("Description"), " "))).show(2) # shows 5 and 3
```

Array Contains

For instance we can see if this array contains a value.

```
%scala
import org.apache.spark.sql.functions.array_contains
df.select(array_contains(split(col("Description"), " "), "WHITE")).
show(2)
```

```
%python
from pyspark.sql.functions import array_contains
df.select(array_contains(split(col("Description"), " "), "WHITE")).
show(2)
```

This is quite powerful because Spark will allow us to manipulate this complex type as another column. We can also query the values of the array with a python-like syntax.

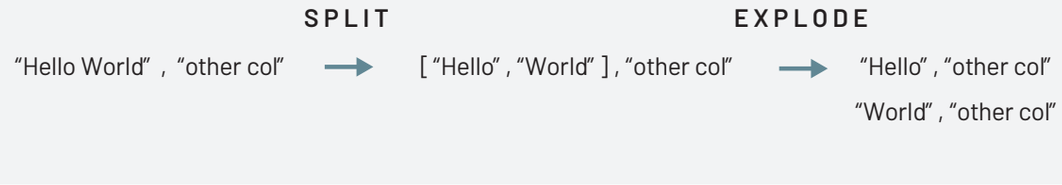
```
%sql
SELECT
   array_contains(split(Description, ' '), 'WHITE')
FROM
   dfTable
   LIMIT 2
```

```
+-------------------------------------------+
|array_contains(split(Description,  ), WHITE)|
+-------------------------------------------+
|                                       true|
|                                       true|
+-------------------------------------------+
```

However this does not solve our current problem. In order to convert a complex type into a set of rows (one per value in our array), we use the explode function.

Explode

The explode function takes a column that consists of arrays and creates one row (with the rest of the values duplicated) per value in the array. The following figure illustrates the process.




```
%scala
import org.apache.spark.sql.functions.{split, explode}
df.withColumn("splitted", split(col("Description"), " "))
   .withColumn("exploded", explode(col("splitted")))
   .select("Description", "InvoiceNo", "exploded")
   .show(2)
```

```
%python
from pyspark.sql.functions import split, explode
df.withColumn("splitted", split(col("Description"), " "))\
   .withColumn("exploded", explode(col("splitted")))\
   .select("Description", "InvoiceNo", "exploded")\
   .show(2)
```

```
%sql
SELECT
   Description,
InvoiceNo,
   exploded
FROM
   (SELECT
      *,
       split(Description, " ") as splitted
   FROM
      dfTable)
   LATERAL VIEW explode(splitted) as exploded
LIMIT 2
```

```
+--------------------+---------+--------+
|         Description|InvoiceNo|exploded|
+--------------------+---------+--------+
|WHITE HANGING HEA...|   536365|   WHITE|
|WHITE HANGING HEA...|   536365| HANGING|
+--------------------+---------+--------+
```

Maps

Maps are used less frequently but are still important to cover. We create them with the map function and key value pairs of columns. Then we can select them just like we might select from an array.

```
%scala
import org.apache.spark.sql.functions.map
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_
map"))
   .selectExpr("complex_map['Description']")
   .show(2)
```

```
%python
from pyspark.sql.functions import create_map
df.select(create_map(col("Description"), col("InvoiceNo")).alias("com-
plex_map"))\
   .show(2)
```

```
%sql
SELECT
   map(Description, InvoiceNo) as complex_map
FROM
   dfTable
WHERE
   Description IS NOT NULL
```

```
+--------------------+
|         complex_map|
+--------------------+
|Map(WHITE HANGING...|
|Map(WHITE METAL L...|
+--------------------+
```

We can query them by using the proper key. A missing key returns null.

```
%scala
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map"))
   .selectExpr("complex_map['WHITE METAL LANTERN']")
   .show(2)
```

```
%python
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map"))\
   .selectExpr("complex_map['WHITE METAL LANTERN']")\
   .show(2)
```

```
+--------------------------------+
|complex_map[WHITE METAL LANTERN]|
+--------------------------------+
|                            null|
|                          536365|
+--------------------------------+
```

We can also explode map types which will turn them into columns.

```
%scala
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_
map"))
   .selectExpr("explode(complex_map)")
   .show(2)
```

```
%python
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_
map"))\
   .selectExpr("explode(complex_map)")\
   .show(2)
```

```
+--------------------+------+
|                 key| value|
+--------------------+------+
|WHITE HANGING HEA...|536365|
| WHITE METAL LANTERN|536365|
+--------------------+------+
```

Working with JSON

Spark has some unique support for working with JSON data. You can operate directly on strings of JSON in Spark and parse from JSON or extract JSON objects. Let's start by creating a JSON column.

```
%scala
val jsonDF = spark.range(1)
   .selectExpr("""
   '{"myJSONKey" : {"myJSONValue" : [1, 2, 3]}}' as jsonString
   """)
```

```
%python
jsonDF = spark.range(1)\
   .selectExpr("""
   '{"myJSONKey" : {"myJSONValue" : [1, 2, 3]}}' as jsonString
   """)
```

We can use the `get_json_object` to inline query a JSON object, be it a dictionary or array. We can use `json_tuple` if this object has only one level of nesting.

```
%scala
import org.apache.spark.sql.functions.{get_json_object, json_tuple}
jsonDF.select(
    get_json_object(col("jsonString"), "$.myJSONKey.myJSONValue[1]"),
    json_tuple(col("jsonString"), "myJSONKey"))
   .show(2)
```

```
%python
from pyspark.sql.functions import get_json_object, json_tuple
jsonDF.select(
    get_json_object(col("jsonString"), "$.myJSONKey.myJSONValue[1]"),
    json_tuple(col("jsonString"), "myJSONKey"))\
   .show(2)
```

The equivalent in SQL would be.

```
jsonDF.selectExpr("json_tuple(jsonString, '$.myJSONKey.myJSONValue[1]')
as res")
```

```
+------+--------------------+
|column|                  c0|
+------+--------------------+
|     2|{"myJSONValue":[1...|
+------+--------------------+
```

We can also turn a StructType into a JSON string using the to_json function.

```
%scala
import org.apache.spark.sql.functions.to_json
df.selectExpr("(InvoiceNo, Description) as myStruct")
   .select(to_json(col("myStruct")))
```

```
%python
from pyspark.sql.functions import to_json
df.selectExpr("(InvoiceNo, Description) as myStruct")\
   .select(to_json(col("myStruct")))
```

This function also accepts a dictionary (map) of parameters that are the same as the JSON data source. We can use the `from_json` function to parse this (or other json) back in. This naturally requires us to specify a schema and optionally we can specify a Map of options as well.

```
%scala
import org.apache.spark.sql.functions.from_json
import org.apache.spark.sql.types._
val parseSchema = new StructType(Array(
    new StructField("InvoiceNo",StringType,true),
    new StructField("Description",StringType,true)))
df.selectExpr("(InvoiceNo, Description) as myStruct")
   .select(to_json(col("myStruct")).alias("newJSON"))
   .select(from_json(col("newJSON"), parseSchema), col("newJSON"))
```

```
%python
from pyspark.sql.functions import from_json
from pyspark.sql.types import *
parseSchema = StructType((
    StructField("InvoiceNo",StringType(),True),
    StructField("Description",StringType(),True)))
df.selectExpr("(InvoiceNo, Description) as myStruct")\
   .select(to_json(col("myStruct")).alias("newJSON"))\
   .select(from_json(col("newJSON"), parseSchema), col("newJSON"))\
```

```
+----------------------+--------------------+
|jsontostructs(newJSON)|             newJSON|
+----------------------+--------------------+
|  [536365,WHITE HAN...|{"InvoiceNo":"536...|
|  [536365,WHITE MET...|{"InvoiceNo":"536...|
+----------------------+--------------------+
```

User-Defined Functions

One of the most powerful things that you can do in Spark is define your own functions. These allow you to write your own custom transformations using Python or Scala and even leverage external libraries like numpy in doing so. These functions are called `user defined functions` or `UDFs` and can take and return one or more columns as input. Spark UDFs are incredibly powerful because they can be written in several different programming languages and do not have to be written in an esoteric format or DSL. They're just functions that operate on the data, record by record. By default, these functions are registered as temporary functions to be used in that specific SparkSession or Context.

While we can write our functions in Scala, Python, or Java, there are performance considerations that you should be aware of. To illustrate this, we're going to walk through exactly what happens when you create UDF, pass that into Spark, and then execute code using that UDF.

The first step is the actual function, we'll just a take a simple one for this example. We'll write a `power3` function that takes a number and raises it to a power of three.

```
%scala
val udfExampleDF = spark.range(5).toDF("num")
def power3(number:Double):Double = {
   number * number * number
}
power3(2.0)
```

```
%python
udfExampleDF = spark.range(5).toDF("num")
def power3(double_value):
   return double_value ** 3
power3(2.0)
```

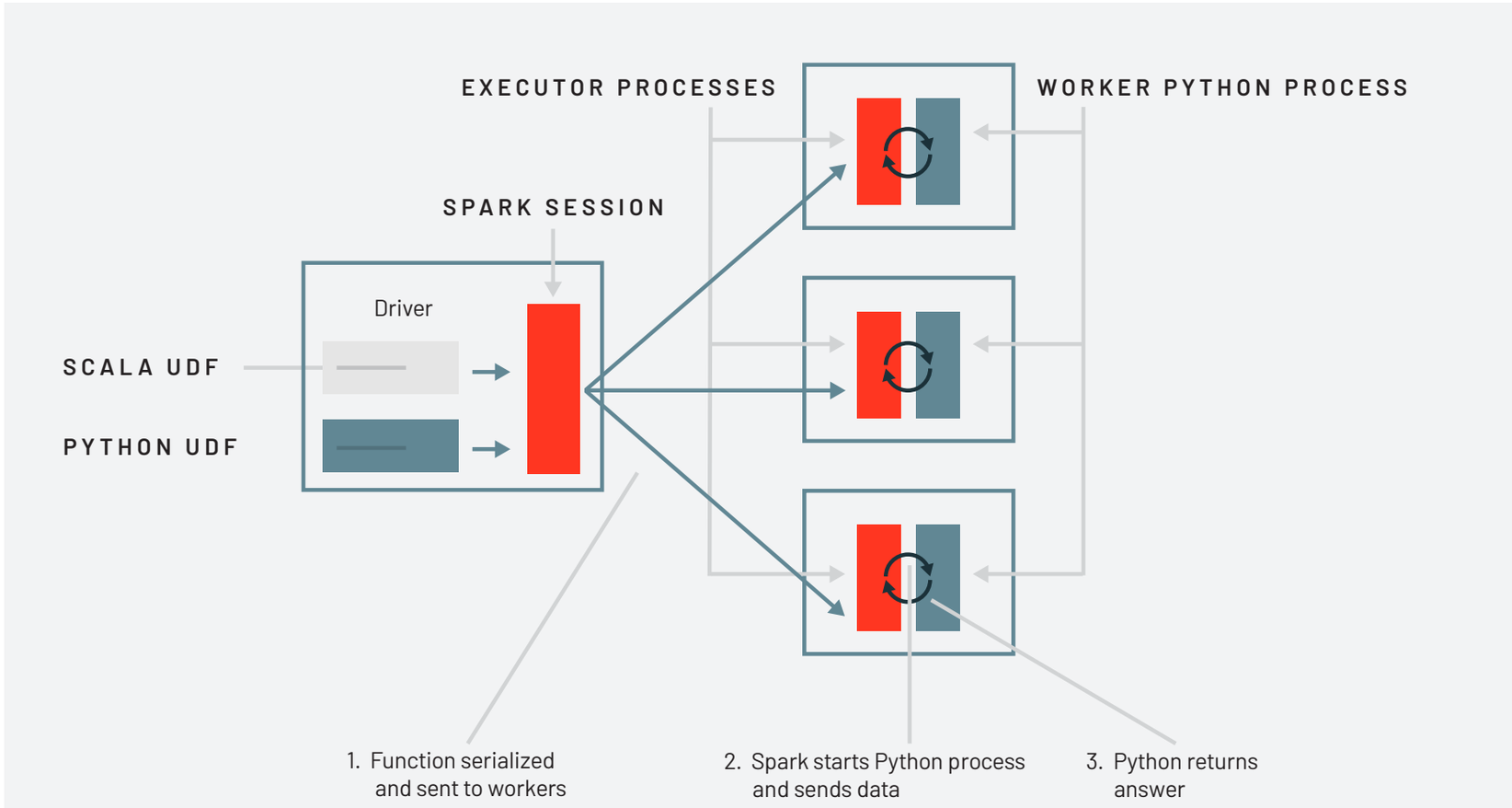
In this trivial example, we can see that our functions work as expected. We are able to provide an individual input and produce the expected result (with this simple test case). Thus far our expectations for the input are high, it must be a specific type and cannot be a null value. See the section in this chapter titled "Working with Nulls in Data".

Now that we've created these functions and tested them, we need to register them with Spark so that we can used them on all of our worker machines. Spark will serialize the function on the driver and transfer it over the network to all executor processes. This happens regardless of language.

Once we go to use the function, there are essentially two different things that occur. If the function is written in Scala or Java then we can use that function within the JVM. This means there will be little performance penalty aside from the fact that we can't take advantage of code generation capabilities that Spark has for built-in functions. There can be performance issues if you create or use a lot of objects which we will cover in the optimization section.

If the function is written in Python, something quite different happens. Spark will start up a python process on the worker, serialize all of the data to a format that python can understand (remember it was in the JVM before), execute the function row by row on that data in the python process, before finally returning the results of the row operations to the JVM and Spark.

WARNING | *Starting up this Python process is expensive but the real cost is in serializing the data to Python. This is costly for two reasons, it is an expensive computation but also once the data enters Python, Spark cannot manage the memory of the worker. This means that you could potentially cause a worker to fail if it becomes resource constrained (because both the JVM and python are competing for memory on the same machine). We recommend that you write your UDFs in Scala - the small amount of time it should take you to write the function in Scala will always yield significant speed ups and on top of that, you can still use the function from Python!*



Now that we have an understanding of the process, let's work through our example. First we need to register the function to be available as a DataFrame function.

```
%scala
import org.apache.spark.sql.functions.udf
val power3udf = udf(power3(_:Double):Double)
```

Now we can use that just like any other DataFrame function.

```
%scala
udfExampleDF.select(power3udf(col("num"))).show()
```

The same applies to Python, we first register it.

```
%python
from pyspark.sql.functions import udf
power3udf = udf(power3)
```

Then we can use it in our DataFrame code.

```
%python
from pyspark.sql.functions import col
udfExampleDF.select(power3udf(col("num"))).show()
```

```
+-----------+
|power3(num)|
+-----------+
|          0|
|          1|
+-----------+
```

Now as of now, we can only use this as DataFrame function. That is to say, we can't use it within a string expression, only on an expression. However, we can also register this UDF as a Spark SQL function. This is valuable because it makes it simple to use this function inside of SQL as well as across languages.

Let's register the function in Scala.

```
%scala
spark.udf.register("power3", power3(_:Double):Double)
udfExampleDF.selectExpr("power3(num)").show(2)
```

Now because this function is registered with Spark SQL, and we've learned that any Spark SQL function or epxression is valid to use as an expression when working with DataFrames, we can turn around and use the UDF that we wrote in Scala, in Python. However rather than using it as a DataFrame function we use it as a SQL expression.

```
%python
udfExampleDF.selectExpr("power3(num)").show(2)
# registered in Scala
```

We can also register our Python function to be available as SQL function and use that in any language as well.

One thing we can also do to make sure that our functions are working correctly is specify a return type. As we saw in the beginning of this section, Spark manages its own type information that does not align exactly with Python's types. Therefore it's a best practice to define the return type for your function when you define it. It is important to note that specifying the return type is not necessary but is a best practice.

If you specify the type that doesn't align with the actual type returned by the function — Spark will not error but rather just return `null` to designate a failure. You can see this if you were to switch the return type in the below function to be a `DoubleType`.

```
%python
from pyspark.sql.types import IntegerType, DoubleType
spark.udf.register("power3py", power3, DoubleType())
```

```
%python
udfExampleDF.selectExpr("power3py(num)").show(2)
# registered via Python
```

This is because the range above creates Integers. When Integers are operated on in Python, Python won't convert them into floats (the corresponding type to Spark's Double type), therefore we see null. We can remedy this by ensuring our Python function returns a float instead of an Integer and the function will behave correctly.

Naturally we can use either of these from SQL too once we register them.

```
%sql
SELECT
   power3py(12), -- doesn't work because of return type
   power3(12)
```

When you want to optionally return a value from a UDF, you should return None in python and an Option type in Scala.

Hive UDFs

As a last note, users can also leverage UDF/UDAF creation via a Hive syntax. To allow for this, first you must enable Hive support when they create their SparkSession (via `SparkSession.builder().enableHiveSupport()`) then you can register UDFs in SQL. This is only supported with pre-compiled Scala and Java packages so you'll have to specify them as a dependency.

```
%sql
CREATE TEMPORARY FUNCTION myFunc AS
   'com.organization.hive.udf.FunctionName'
```

Additionally, you can register this as a permanent function in the Hive Metastore by removing `TEMPORARY`.

CHAPTER 3: Delta Lake Quickstart

Delta Lake is an open-source storage layer that brings data reliability to data lakes. Delta Lake provides ACID transactions, can handle metadata at scale, and can unify streaming and batch data processing. Delta Lake runs on top of your existing data lake and is fully compatible with Apache Spark APIs.

How to start using Delta Lake

The Delta Lake package is available as with the—packages option. In our example, we will also demonstrate the ability to VACUUM files and execute Delta Lake SQL commands within Apache Spark. As this is a short demonstration, we will also enable the following configurations:

- `spark.databricks.delta.retentionDurationCheck.enabled=false` to allow us to vacuum files shorter than the default retention duration of 7 days. Note, this is only required for the SQL command VACUUM.
- `spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension` to enable Delta Lake SQL commands within Apache Spark; this is not required for Python or Scala API calls.

```
# Using Spark Packages
./bin/pyspark --packages io.delta:delta-core_2.11:0.4.0 --conf “spark.databricks.
delta.retentionDurationCheck.enabled=false” --conf “spark.sql.extensions=io.delta.sql.
DeltaSparkSessionExtension”
```

Loading and saving our Delta Lake data

This scenario will be using the On-time flight performance or Departure Delays dataset generated from the RITA BTS Flight Departure Statistics; some examples of this data in action include the 2014 Flight Departure Performance via d3.js Crossfilter and On-Time Flight Performance with GraphFrames for Apache Spark™. This dataset can be downloaded locally from this github location. Within pyspark, start by reading the dataset.

```
# Location variables
tripdelaysFilePath = "/root/data/departuredelays.csv"
pathToEventsTable = "/root/deltalake/departureDelays.delta"

# Read flight delay data
departureDelays = spark.read \
.option("header", "true") \
.option("inferSchema", "true") \
.csv(tripdelaysFilePath)
```

Next, let's save our *departureDelays* dataset to a Delta Lake table. By saving this table to Delta Lake storage, we will be able to take advantage of its features including ACID transactions, unified batch and streaming, and time travel.

```
# Save flight delay data into Delta Lake format
departureDelays \
.write \
.format("delta") \
.mode("overwrite") \
.save("departureDelays.delta")
```

NOTE | This approach is similar to how you would normally save Parquet data; instead of specifying `format("parquet")`, you will now specify `format("delta")`. If you were to take a look at the underlying file system, you will notice four files created for the *departureDelays* Delta Lake table.

```
/departureDelays.delta$ ls -l
.
..
_delta_log
part-00000-df6f69ea-e6aa-424b-bc0e-f3674c4f1906-c000.snappy.parquet
part-00001-711bcce3-fe9e-466e-a22c-8256f8b54930-c000.snappy.parquet
part-00002-778ba97d-89b8-4942-a495-5f6238830b68-c000.snappy.parquet
part-00003-1a791c4a-6f11-49a8-8837-8093a3220581-c000.snappy.parquet
```

NOTE | *The `_delta_log` is the folder that contains the Delta Lake transaction log. For more information, refer to Diving Into Delta Lake: Unpacking The Transaction Log.*

Now, let's reload the data but this time our DataFrame will be backed by Delta Lake.

```
# Load flight delay data in Delta Lake format
delays_delta = spark \
.read \
.format("delta") \
.load("departureDelays.delta")
# Create temporary view
delays_delta.createOrReplaceTempView("delays_delta")
# How many flights are between Seattle and San Francisco
park.sql("select count(1) from delays_delta where origin = 'SEA' and destination = 'SFO'").show()
```

	count(1)
0	1698

Finally, let's determine the number of flights originating from Seattle to San Francisco; in this dataset, there are 1698 flights.

In-place Conversion to Delta Lake

If you have existing Parquet tables, you have the ability to perform in-place conversions your tables to Delta Lake thus not needing to rewrite your table. To convert the table, you can run the following commands.

```
from delta.tables import *
# Convert non partitioned parquet table at path '/path/to/table'
deltaTable = DeltaTable.convertToDelta(spark, "parquet.`/path/to/table`")
# Convert partitioned parquet table at path '/path/to/table' and partitioned by integer column named 'part' partitionedDeltaTable = DeltaTable.convert-
ToDelta(spark, "parquet.`/path/to/table`", "part int")
park.sql("select count(1) from delays_delta where origin = 'SEA' and destination = 'SFO'").show()
```

For more information, including how to do this conversion in Scala and SQL, refer to Convert to Delta Lake.

Delete our Flight Data

To delete data from your traditional Data Lake table, you will need to:

1. Select all of the data from your table not including the rows you want to delete
2. Create a new table based on the previous query
3. Delete the original table
4. Rename the new table to the original table name for downstream dependencies.

Instead of performing all of these steps, with Delta Lake, we can simplify this process by running a DELETE statement. To show this, let's delete all of the flights that had arrived early or on-time (i.e. `delay < 0`).

```
from delta.tables import *
from pyspark.sql.functions import *
# Access the Delta Lake table
deltaTable = DeltaTable.forPath(spark, pathToEventsTable
)
# Delete all on-time and early flights
deltaTable.delete("delay < 0")
# How many flights are between Seattle and San Francisco
spark.sql("select count(1) from delays_delta where origin = 'SEA' and
destination = 'SFO'").show()
```

	count(1)
0	837

After we *delete* (more on this below) all of the on-time and early flights, as you can see from the preceding query there are 837 late flights originating from Seattle to San Francisco. If you review the file system, you will notice there are more files even though you *deleted* data.

```
/departureDelays.delta$ ls -l
_delta_log
part-00000-a2a19ba4-17e9-4931-9bbf-3c9d4997780b-c000.snappy.parquet
part-00000-df6f69ea-e6aa-424b-bc0e-f3674c4f1906-c000.snappy.parquet
part-00001-711bcce3-fe9e-466e-a22c-8256f8b54930-c000.snappy.parquet
part-00001-a0423a18-62eb-46b3-a82f-ca9aac1f1e93-c000.snappy.parquet
part-00002-778ba97d-89b8-4942-a495-5f6238830b68-c000.snappy.parquet
part-00002-bfaa0a2a-0a31-4abf-aa63-162402f802cc-c000.snappy.parquet
part-00003-1a791c4a-6f11-49a8-8837-8093a3220581-c000.snappy.parquet
part-00003-b0247e1d-f5ce-4b45-91cd-16413c784a66-c000.snappy.parquet
```

In traditional data lakes, *deletes* are performed by re-writing the entire table excluding the values to be deleted. With Delta Lake, *deletes* instead are performed by selectively writing new versions of the files containing the data be deleted and only *marks* the previous files as deleted. This is because Delta Lake uses multiversion concurrency control to do atomic operations on the table: for example, while one user is deleting data, another user may be querying the previous version of the table. This multi-version model also enables us to travel back in time (i.e. time travel) and query previous versions as we will see later.

Update our Flight Data

To update data from your traditional Data Lake table, you will need to:

1. Select all of the data from your table not including the rows you want to modify
2. Modify the rows that need to be updated/changed
3. Merge these two tables to create a new table
4. Delete the original table
5. Rename the new table to the original table name for downstream dependencies.

Instead of performing all of these steps, with Delta Lake, we can simplify this process by running an UPDATE statement. To show this, let's update all of the flights originating from Detroit to Seattle.

```
Update all flights originating from Detroit to now be originating from Seattle
deltaTable.update("origin = 'DTW'", { "origin": "'SEA'" } )
# How many flights are between Seattle and San Francisco
spark.sql("select count(1) from delays_delta where origin = 'SEA' and destination = 'SFO'").show()
```

	count(1)
0	986

With the Detroit flights now tagged as Seattle flights, we now have 986 flights originating from Seattle to San Francisco. If you were to list the file system for your *departureDelays* folder (i.e. `$../departureDelays/ls -l`), you will notice there are now 11 files (instead of the 8 right after deleting the files and the four files after creating the table)

Merge our Flight Data

A common scenario when working with a data lake is to continuously append data to your table. This often results in duplicate data (rows you do not want inserted into your table again), new rows that need to be inserted, and some rows that need to be updated. With Delta Lake, all of this can be achieved by using the merge operation (similar to the SQL MERGE statement).

Let's start with a sample dataset that you will want to be updated, inserted, or deduplicated with the following query.

```
# What flights between SEA and SFO for these date periods
spark.sql("select * from delays_delta where origin = 'SEA' and destination = 'SFO' and date like '1010%' limit 10").show()
```

	date	delay	distance	origin	destination
0	1010521	0	590	SEA	SFO
1	1010710	31	590	SEA	SFO
2	1010730	5	590	SEA	SFO
3	1010955	104	590	SEA	SFO

The output of this query looks like the table at left. Note, the color-coding has been added to this blog to clearly identify which rows are deduplicated (blue), updated (yellow), and inserted (green).

Next, let's generate our own `merge_table` that contains data we will insert, update or de-duplicate with the following code snippet.

```
items = [(1010710, 31, 590, 'SEA', 'SFO'), (1010521, 10, 590, 'SEA', 'SFO'), (1010822, 31, 590, 'SEA', 'SFO')]
cols = ['date', 'delay', 'distance', 'origin', 'destination']
merge_table = spark.createDataFrame(items, cols)
merge_table.toPandas()
```

	date	delay	distance	origin	destination
0	1010521	10	590	SEA	SFO
1	1010710	31	590	SEA	SFO
2	1010832	31	590	SEA	SFO

In the preceding table (`merge_table`), there are three rows that with a unique date value:

1. 1010521: this row needs to update the *flights* table with a new delay value (yellow)
2. 1010710: this row is a *duplicate* (blue)
3. 1010832: this is a new row to be *inserted* (green)

With Delta Lake, this can be easily achieved via a merge statement as noted in the following code snippet.

```
# Merge merge_table with flights
deltaTable.alias("flights") \
    .merge(merge_table.alias("updates"),"flights.date = updates.date") \
    .whenMatchedUpdate(set = { "delay" : "updates.delay" } ) \
    .whenNotMatchedInsertAll() \
    .execute()
# What flights between SEA and SFO for these date periods
spark.sql("select * from delays_delta where origin = 'SEA' and destination = 'SFO' and date like '1010%' limit 10").show()
```

All three actions of de-duplication, update, and insert was efficiently completed with one statement.

	date	delay	distance	origin	destination
0	1010521	10	590	SEA	SFO
1	1010710	31	590	SEA	SFO
2	1010730	5	590	SEA	SFO
3	1010832	31	590	SEA	SFO

View Table History

As previously noted, after each of our transactions (delete, update), there were more files created within the file system. This is because for each transaction, there are different versions of the Delta Lake table. This can be seen by using the `DeltaTable.history()` method as noted below.

```
deltaTable.history().show()
+-------+-------------------+------+--------+---------+--------------------+----+--------+---------+-----------+--------------+-------------+
|version|          timestamp|userId|userName|operation| operationParameters| job|notebook|clusterId|readVersion|isolationLevel|isBlindAppend|
+-------+-------------------+------+--------+---------+--------------------+----+--------+---------+-----------+--------------+-------------+
|      2|2019-09-29 15:41:22|  null|    null|   UPDATE|[predicate -> (or...|null|    null|     null|          1|          null|        false|
|      1|2019-09-29 15:40:45|  null|    null|   DELETE|[predicate -> ["(...|null|    null|     null|          0|          null|        false|
|      0|2019-09-29 15:40:14|  null|    null|    WRITE|[mode -> Overwrit...|null|    null|     null|       null|          null|        false|
+-------+-------------------+------+--------+---------+--------------------+----+--------+---------+-----------+--------------+-------------+
```

NOTE | *You can also perform the same task with SQL:* `spark.sql("DESCRIBE HISTORY '" + pathToEventsTable + "'").show()`

As you can see, there are three rows representing the different versions of the table (below is an abridged version to help make it easier to read) for each of the operations (create table, delete, and update):

version	timestamp	operation	operationParameters
2	2019-09-29 15:41:22	UPDATE	[predicate -> (or…
1	2019-09-29 15:40:45	DELETE	[predicate -> ["(…
0	2019-09-29 15:40:14	WRITE	[mode -> Overwrit…

Travel Back in Time with Table History

With Time Travel, you can see review the Delta Lake table as of the version or timestamp. For more information, refer to Delta Lake documentation > Read older versions of data using Time Travel. To view historical data, specify the version or Timestamp option; in the code snippet below, we will specify the version option

```
# Load DataFrames for each version
dfv0 = spark.read.format("delta").option("versionAsOf", 0).load("departureDelays.delta")
dfv1 = spark.read.format("delta").option("versionAsOf", 1).load("departureDelays.delta")
dfv2 = spark.read.format("delta").option("versionAsOf", 2).load("departureDelays.delta")
# Calculate the SEA to SFO flight counts for each version of history
cnt0 = dfv0.where("origin = 'SEA'").where("destination = 'SFO'").count()
cnt1 = dfv1.where("origin = 'SEA'").where("destination = 'SFO'").count()
cnt2 = dfv2.where("origin = 'SEA'").where("destination = 'SFO'").count()
# Print out the value
print("SEA -> SFO Counts: Create Table: %s, Delete: %s, Update: %s" % (cnt0, cnt1, cnt2))
## Output
SEA -> SFO Counts: Create Table: 1698, Delete: 837, Update: 986
```

Whether for governance, risk management, and compliance (GRC) or rolling back errors, the Delta Lake table contains both the metadata (e.g. recording the fact that a delete had occurred with these operators) and data (e.g. the actual rows deleted). But how do we remove the data files either for compliance or size reasons?

Cleanup Old Table Versions with Vacuum

The Delta Lake vacuum method will delete all of the rows (and files) by default that are older than 7 days (reference: Delta Lake Vacuum). If you were to view the file system, you'll notice the 11 files for your table.

```
/departureDelays.delta$ ls -l
_delta_log
part-00000-5e52736b-0e63-48f3-8d56-50f7cfa0494d-c000.snappy.parquet
part-00000-69eb53d5-34b4-408f-a7e4-86e000428c37-c000.snappy.parquet
part-00000-f8edaf04-712e-4ac4-8b42-368d0bbdb95b-c000.snappy.parquet
part-00001-20893eed-9d4f-4c1f-b619-3e6ea1fdd05f-c000.snappy.parquet
part-00001-9b68b9f6-bad3-434f-9498-f92dc4f503e3-c000.snappy.parquet
part-00001-d4823d2e-8f9d-42e3-918d-4060969e5844-c000.snappy.parquet
part-00002-24da7f4e-7e8d-40d1-b664-95bf93ffeadb-c000.snappy.parquet
part-00002-3027786c-20a9-4b19-868d-dc7586c275d4-c000.snappy.parquet
part-00002-f2609f27-3478-4bf9-aeb7-2c78a05e6ec1-c000.snappy.parquet
part-00003-850436a6-c4dd-4535-a1c0-5dc0f01d3d55-c000.snappy.parquet
part-00003-b9292122-99a7-4223-aaa9-8646c281f199-c000.snappy.parquet
```

To delete all of the files so that you only keep the current snapshot of data, you will specify a small value for the vacuum method (instead of the default retention of 7 days).

```
# Remove all files older than 0 hours old.
deltaTable.vacuum(0)
```

NOTE | *You perform the same task via SQL syntax:*

```
# Remove all files older than 0 hours old
spark.sql("VACUUM '" + pathToEventsTable + "' RETAIN 0 HOURS")
```

Once the vacuum has completed, when you review the file system you will notice fewer files as the historical data has been removed.

```
/departureDelays.delta$ ls -l
_delta_log
part-00000-f8edaf04-712e-4ac4-8b42-368d0bbdb95b-c000.snappy.parquet
part-00001-9b68b9f6-bad3-434f-9498-f92dc4f503e3-c000.snappy.parquet
part-00002-24da7f4e-7e8d-40d1-b664-95bf93ffeadb-c000.snappy.parquet
part-00003-b9292122-99a7-4223-aaa9-8646c281f199-c000.snappy.parquet
```

NOTE | *The ability to time travel back to a version older than the retention period is lost after running vacuum.*

What's Next

Try out Delta Lake today by trying out the preceding code snippets on your Apache Spark 2.4.3 (or greater) instance. By using Delta Lake, you can make your data lakes more reliable (whether you create a new one or migrate an existing data lake). To learn more, refer to delta.io and join the Delta Lake community via Slack and Google Group. You can track all the upcoming releases and planned features in github milestones.

The datasets used in the book are also available for you to explore:

SPARK: THE DEFINITIVE GUIDE DATASETS

Get started with a free trial of Databricks and take your big data and data science projects to the next level today.

START YOUR FREE TRIAL

Contact us for a personalized demo

CONTACT