Chapter 6. Working with Different Types of Data

Chapter 5 presented basic DataFrame concepts and abstractions. This chapter covers building expressions, which are the bread and butter of Spark's structured operations. We also review working with a variety of different kinds of data, including the following:

- Booleans
- Numbers
- Strings
- Dates and timestamps
- Handling null
- Complex types
- User-defined functions

Where to Look for APIs

Before we begin, it's worth explaining where you as a user should look for transformations. Spark is a growing project, and any book (including this one) is a snapshot in time. One of our priorities in this book is to teach where, as of this writing, you should look to find functions to transform your data. Following are the key places to look:

`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, which are available at this link. (http://bit.ly/2rKkALY)

`Dataset` submodules like `DataFrameStatFunctions` (http://bit.ly/2DPYhJC) and `DataFrameNaFunctions` (http://bit.ly/2DPAqd3) have more methods that solve specific sets of problems. `DataFrameStatFunctions`, for example, holds a variety of statistically related functions, whereas `DataFrameNaFunctions` refers to functions that are relevant when working with null data.

`Column` Methods

These were introduced for the most part in Chapter 5. They hold a variety of general column-related methods like `alias` or `contains`. You can find the API Reference for Column methods here (http://bit.ly/2FloFbr).

`org.apache.spark.sql.functions` contains a variety of functions for a range of different data types. Often, you'll see the entire package imported because they are used so frequently. You can find SQL and DataFrame functions here. (http://bit.ly/2DPAycx)

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 might create more rows or reduce the number of rows available. To begin, let's read in the `DataFrame` that we'll be using for this analysis:

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

```
# in Python
df = spark.read.format("csv")\
  .option("header", "true")\
  .option("inferSchema", "true")\
  .load("/data/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|        Invoic
+---------+---------+--------------------+--------+--------------
|   536365|   85123A|WHITE HANGING HEA...|       6|2010-12-01 08:
|   536365|    71053| WHITE METAL LANTERN|       6|2010-12-01 08:
...
|   536367|    21755|LOVE BUILDING BLO...|       3|2010-12-01 08:
|   536367|    21777|RECIPE BOX WITH M...|       4|2010-12-01 08:
+---------+---------+--------------------+--------+--------------
```

Converting to Spark Types

One thing you'll see us do throughout this chapter is convert native types to Spark types. We do this by using the first function that we introduce here, the `lit` function. This function converts a type in another language to its correspnding Spark representation. Here's how we can convert a couple of different kinds of Scala and Python values to their respective Spark types:

```
// in Scala
import org.apache.spark.sql.functions.lit
df.select(lit(5), lit("five"), lit(5.0))
```

```
# in 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 use the values directly:

```
-- in SQL
SELECT 5, "five", 5.0
```

Working with Booleans

Booleans are essential 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 for when a row of data must either pass the 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-than or greater-than:

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

WARNING

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

```
// in 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:

```
# in 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 you access to another way of expressing "does not equal":

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

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

We mentioned that you can specify Boolean expressions with multiple parts when you 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 statements 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. Although you can specify your statements explicitly by using `and` if you like, they're often easier to understand and to read if you specify them serially. `or` statements need to be specified in the same statement:

```
// in Scala
val priceFilter = col("UnitPrice") > 600
val descripFilter = col("Description").contains("POSTAGE")
df.where(col("StockCode").isin("DOT")).where(priceFilter.or(descripFilt
  .show()
```

```
# in 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).
```

```
-- in SQL
SELECT * FROM dfTable WHERE StockCode in ("DOT") AND(UnitPrice > 600 C
    instr(Description, "POSTAGE") >= 1)
```

```
+---------+---------+--------------+--------+-------------------+
|InvoiceNo|StockCode|   Description|Quantity|        InvoiceDate|
+---------+---------+--------------+--------+-------------------+
|   536544|      DOT|DOTCOM POSTAGE|       1|2010-12-01 14:32:00|
|   536592|      DOT|DOTCOM POSTAGE|       1|2010-12-01 17:06:00|
+---------+---------+--------------+--------+-------------------+
```

Boolean expressions are not just reserved to filters. To filter a `DataFrame`, you can also just specify a Boolean column:

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

```
# in 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 | descripFil
  .where("isExpensive")\
  .select("unitPrice", "isExpensive").show(5)
```

```
-- in SQL
SELECT UnitPrice, (StockCode = 'DOT' AND
  (UnitPrice > 600 OR instr(Description, "POSTAGE") >= 1)) as isExpens
FROM dfTable
WHERE (StockCode = 'DOT' AND
       (UnitPrice > 600 OR instr(Description, "POSTAGE") >= 1))
```

Notice how we did not need 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 following two statements are equivalent:

```
// in 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:

```
# in 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 if you're working with null data when creating Boolean expressions. If there is a null in your data, you'll need to treat things a bit differently. Here's how you can ensure that you perform a null-safe equivalence test:

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

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

Working with Numbers

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 that we're working with numerical data types.

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

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

```
# in Python
from pyspark.sql.functions import expr, pow
fabricatedQuantity = pow(col("Quantity") * col("UnitPrice"), 2) + 5
df.select(expr("CustomerId"), fabricatedQuantity.alias("realQuantity")
+----------+------------------+
|CustomerId|      realQuantity|
+----------+------------------+
|   17850.0|239.08999999999997|
|   17850.0|          418.7156|
+----------+------------------+
```

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 as a SQL expression, as well:

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

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

```
-- in SQL
SELECT customerId, (POWER((Quantity * UnitPrice), 2.0) + 5) as realQu
FROM dfTable
```

Another common numerical task is rounding. If you'd like to just round to a whole number, oftentimes you can cast the value 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 the following example, we round to one decimal place:

```
// in Scala
import org.apache.spark.sql.functions.{round, bround}
df.select(round(col("UnitPrice"), 1).alias("rounded"), col("UnitPrice")
```

By default, the `round` function rounds up if you're exactly in between two numbers. You can round down by using the `bround`:

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

```
# in Python
from pyspark.sql.functions import lit, round, bround

df.select(round(lit("2.5")), bround(lit("2.5"))).show(2)
```

```
-- in 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:

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

```
# in Python
from pyspark.sql.functions import corr
df.stat.corr("Quantity", "UnitPrice")
df.select(corr("Quantity", "UnitPrice")).show()
```

```
-- in SQL
SELECT corr(Quantity, UnitPrice) FROM dfTable
```

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

Another 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. You should use this primarily for viewing in the console because the schema might change in the future:

```
// in Scala
df.describe().show()
```

```
# in 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:

```
// in Scala
import org.apache.spark.sql.functions.{count, mean, stddev_pop, min, ma
```

```
# in Python
from pyspark.sql.functions import count, mean, stddev_pop, min, max
```

There are a number of statistical functions available in the StatFunctions Package (accessible using `stat` as we see in the code block below). These are DataFrame methods that you can use to calculate a variety of different things. For instance, you can calculate either exact or approximate quantiles of your data using the `approxQuantile` method:

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

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

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

```
// in Scala
df.stat.crosstab("StockCode", "Quantity").show()
```

```
# in Python
df.stat.crosstab("StockCode", "Quantity").show()
```

```
// in Scala
df.stat.freqItems(Seq("StockCode", "Quantity")).show()
```

```
# in Python
df.stat.freqItems(["StockCode", "Quantity"]).show()
```

As a last note, we can also add a unique ID to each row by using the function `monotonically_increasing_id`. This function generates a unique value for each row, starting with 0:

```
// in Scala
import org.apache.spark.sql.functions.monotonically_increasing_id
df.select(monotonically_increasing_id()).show(2)
```

```
# in Python
from pyspark.sql.functions import monotonically_increasing_id
df.select(monotonically_increasing_id()).show(2)
```

There are functions added with every release, so check the documentation for more methods. For instance, there are some random data generation tools (e.g., `rand()`, `randn()`) with which you can randomly generate data; however, there are potential determinism issues when doing so. (You can find discussions about these challenges on the Spark mailing list.) There are also a number of more advanced tasks like bloom filtering and sketching algorithms available in the stat package 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 it's worth explaining what you can do with strings. You might be manipulating log files performing regular expression extraction or substitution, or checking for simple string existence, or making all strings uppercase or lowercase.

Let's begin with the last task because it's the most straightforward. The `initcap` function will capitalize every word in a given string when that word is separated from another by a space.

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

```
# in Python
from pyspark.sql.functions import initcap
df.select(initcap(col("Description"))).show()
```

```
-- in SQL
SELECT initcap(Description) FROM dfTable
```

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

As just mentioned, you can cast strings in uppercase and lowercase, as well:

```
// in Scala
import org.apache.spark.sql.functions.{lower, upper}
df.select(col("Description"),
  lower(col("Description")),
  upper(lower(col("Description")))).show(2)
```

```
# in Python
from pyspark.sql.functions import lower, upper
df.select(col("Description"),
    lower(col("Description")),
    upper(lower(col("Description")))).show(2)
```

```
-- in SQL
SELECT Description, lower(Description), Upper(lower(Description)) FROM
```

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

Another trivial task is adding or removing spaces around a string. You can do this by using `lpad`, `ltrim`, `rpad` and `rtrim`, `trim`:

```
// in Scala
import org.apache.spark.sql.functions.{lit, ltrim, rtrim, rpad, lpad, t
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)
```

```
# in 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)
```

```
-- in 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     |
+---------+---------+-----+---+----------+
```

Note 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 in another or replacing all mentions of a string with another value. This is often done with a tool called *regular expressions* that exists 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 takes advantage of the complete power of Java regular expressions. The Java regular expression 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 in order 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 color names in our description column:

```
// in 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
  col("Description")).show(2)
```

```
# in 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("col
  col("Description")).show(2)
```

```
-- in SQL
SELECT
  regexp_replace(Description, 'BLACK|WHITE|RED|GREEN|BLUE', 'COLOR') a
  color_clean, Description
FROM dfTable
```

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

Another task might be to replace given characters with other characters. Building this as a 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:

```
// in Scala
import org.apache.spark.sql.functions.translate
df.select(translate(col("Description"), "LEET", "1337"), col("Descripti
  .show(2)
```

```
# in Python
from pyspark.sql.functions import translate
df.select(translate(col("Description"), "LEET", "1337"),col("Descripti
  .show(2)
```

```
-- in SQL
SELECT translate(Description, 'LEET', '1337'), Description FROM dfTabl
```

```
+----------------------------------+--------------------+
|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:

```
// in 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_c
     col("Description")).show(2)
```

```
# in 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_c
     col("Description")).show(2)
```

```
-- in SQL
SELECT regexp_extract(Description, '(BLACK|WHITE|RED|GREEN|BLUE)', 1),
  Description
FROM dfTable
```

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

Sometimes, rather than extracting values, we simply want to check for their existence. We can do this with the `contains` method on each column. This will return a Boolean declaring whether the value you specify is in the column’s string:

```
// in Scala
val containsBlack = col("Description").contains("BLACK")
val containsWhite = col("DESCRIPTION").contains("WHITE")
df.withColumn("hasSimpleColor", containsBlack.or(containsWhite))
  .where("hasSimpleColor")
  .select("Description").show(3, false)
```

In Python and SQL, we can use the `instr` function:

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

```
-- in SQL
```

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

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

This is trivial with just two values, but it becomes more complicated when there are values.

Let's work through this in a more rigorous 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`. Using this feature, we can effectively unravel an array of arbitrary length and pass it as arguments to a function. This, coupled with `select` makes it possible for us to create arbitrary numbers of columns dynamically:

```
// in 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 the same basic feature:

```
# in Python
from pyspark.sql.functions import expr, locate
simpleColors = ["black", "white", "red", "green", "blue"]
def color_locator(column, color_string):
  return locate(color_string.upper(), column)\
          .cast("boolean")\
          .alias("is_" + c)
selectedColumns = [color_locator(df.Description, c) for c in simpleCol
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 can often help you programmatically generate columns or Boolean filters in a way that is simple to understand and extend. We could extend this to calculating the smallest common denominator for a given input value, or whether 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 ensure 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, which 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.

As we hinted earlier, 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 version 2.1 and before, Spark parsed 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.

```
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)
```

Although Spark will do read dates or times on a best-effort basis. However, sometimes there will be no getting around working with strangely formatted dates and times. The key to understanding 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` class supports only second-level precision, which means that if you're going to be working with milliseconds or microseconds, you'll need 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 ensure that 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 begin with the basics and get the current date and the current timestamps:

```
// in 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")
```

```
# in 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 five days from today. These functions take a column and then the number of days to either add or subtract as the arguments:

```
// in Scala
import org.apache.spark.sql.functions.{date_add, date_sub}
dateDF.select(date_sub(col("today"), 5), date_add(col("today"), 5)).sho
```

```
# in Python
from pyspark.sql.functions import date_add, date_sub
dateDF.select(date_sub(col("today"), 5), date_add(col("today"), 5)).sh
```

```
-- in 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, and because the number of days varies from month to month, there also exists a function, `months_between`, that gives you the number of months between two dates:

```
// in Scala
import org.apache.spark.sql.functions.{datediff, months_between, to_dat
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)
```

```
# in 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)
```

```
-- in 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|
+--------------------------+
```

Notice that we introduced a new function: 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 (http://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html) which will be important to reference if you use this function:

```
// in 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)
```

```
# in 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)
```

Spark will not throw an error if it cannot parse the date; rather, it will just return `null`. This can be a bit tricky in larger pipelines because you might 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"))).sh
```

```
+-------------------+-------------------+
|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 might match the correct format, whereas others do not. In the previous example, notice how the second date appears as Decembers 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 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.

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

```
// in Scala
import org.apache.spark.sql.functions.to_date
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")
```

```
# in Python
from pyspark.sql.functions import to_date
dateFormat = "yyyy-dd-MM"
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")
```

```
-- in SQL
SELECT to_date(date, 'yyyy-dd-MM'), to_date(date2, 'yyyy-dd-MM'), to_d
FROM dateTable2
```

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

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

```
// in Scala
import org.apache.spark.sql.functions.to_timestamp
cleanDateDF.select(to_timestamp(col("date"), dateFormat)).show()
```

```
# in Python
from pyspark.sql.functions import to_timestamp
cleanDateDF.select(to_timestamp(col("date"), dateFormat)).show()
```

```
-- in SQL
SELECT to_timestamp(date, 'yyyy-dd-MM'), to_timestamp(date2, 'yyyy-dd-
FROM dateTable2
```

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

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

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

After we have our date or timestamp in 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 conversions.

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. For more information, see Chapter 5 (where we discuss ordering), and also refer back to “Working with Booleans”.

WARNING

Nulls are a challenging part of all programming, and Spark is no exception. In our opinion, 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 in which 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 simply 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 difficult 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 non-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:

```
// in Scala
import org.apache.spark.sql.functions.coalesce
df.select(coalesce(col("Description"), col("CustomerId"))).show()
```

```
# in Python
from pyspark.sql.functions import coalesce
df.select(coalesce(col("Description"), col("CustomerId"))).show()
```

ifnull, nullIf, nvl, and nvl2

There are several other SQL functions that you can use to achieve similar things. `ifnull` allows you to select the second value if the first is null, and defaults to the first. Alternatively, you could use `nullif`, which returns null if the two values are equal or else returns the second if they are not. `nvl` returns the second value if the first is null, but defaults to the first. Finally, `nvl2` returns the second value if the first is not null; otherwise, it will return the last specified value (`else_value` in the following example):

```
-- in 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 function is `drop`, which removes rows that contain nulls. The default is to drop any row in which any value is null:

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

In SQL, we have to do this column by column:

```
-- in SQL
SELECT * FROM dfTable WHERE Description IS NOT NULL
```

Specifying `"any"` as an argument drops a row if any of the values are null. Using "all" drops the row only 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:

```
// in Scala
df.na.drop("all", Seq("StockCode", "InvoiceNo"))
```

```
# in Python
df.na.drop("all", subset=["StockCode", "InvoiceNo"])
```

fill

Using the `fill` function, you can fill one or more columns with a set of values. This can be done by specifying a map—that is a particular value and a set of columns.

For example, to fill all `null` values in columns of type String, you might specify the following:

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

We could do the same for columns of type Integer by using `df.na.fill(5:Integer)`, or for Doubles `df.na.fill(5:Double)`. To specify columns, we just pass in an array of column names like we did in the previous example:

```
// in Scala
df.na.fill(5, Seq("StockCode", "InvoiceNo"))
```

```
# in Python
df.na.fill("all", subset=["StockCode", "InvoiceNo"])
```

We can also do this 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:

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

```
# in 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 you 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:

```
// in Scala
df.na.replace("Description", Map("" -> "UNKNOWN"))
```

```
# in Python
df.na.replace([""], ["UNKNOWN"], "Description")
```

Ordering

As we discussed in Chapter 5, you can use `asc_nulls_first`, `desc_nulls_first`, `asc_nulls_last`, or `desc_nulls_last` to specify where you would like your 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 that 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", "*")
```

```
// in Scala
import org.apache.spark.sql.functions.struct
val complexDF = df.select(struct("Description", "InvoiceNo").alias("com
complexDF.createOrReplaceTempView("complexDF")
```

```
# in Python
from pyspark.sql.functions import struct
complexDF = df.select(struct("Description", "InvoiceNo").alias("comple
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 by using *. This brings up all the columns to the top-level DataFrame:

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

```
-- in SQL
SELECT complex.* FROM complexDF
```

Arrays

To define arrays, let’s work through a use case. With our current data, our objective 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 by using the `split` function and specify the delimiter:

```
// in Scala
import org.apache.spark.sql.functions.split
df.select(split(col("Description"), " ")).show(2)
```

```
# in Python
from pyspark.sql.functions import split
df.select(split(col("Description"), " ")).show(2)
```

```
-- in SQL
SELECT split(Description, ' ') FROM dfTable
```

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

This is quite powerful because Spark allows us to manipulate this complex type as another column. We can also query the values of the array using Python-like syntax:

```
// in Scala
df.select(split(col("Description"), " ").alias("array_col"))
  .selectExpr("array_col[0]").show(2)
```

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

```
-- in SQL
SELECT split(Description, ' ')[0] FROM dfTable
```

This gives us the following result:

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

Array Length

We can determine the array’s length by querying for its size:

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

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

array_contains

We can also see whether this array contains a value:

```
// in Scala
import org.apache.spark.sql.functions.array_contains
df.select(array_contains(split(col("Description"), " "), "WHITE")).show
```

```
# in Python
from pyspark.sql.functions import array_contains
df.select(array_contains(split(col("Description"), " "), "WHITE")).sho
```

```
-- in SQL
SELECT array_contains(split(Description, ' '), 'WHITE') FROM dfTable
```

This gives us the following result:

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

However, this does not solve our current problem. To convert a complex type into a set of rows (one per value in our array), we need to 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. Figure 6-1 illustrates the process.

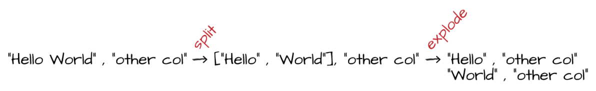

Figure 6-1. Exploding a column of text

```
// in 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)
```

```
# in 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)
```

```
-- in SQL
SELECT Description, InvoiceNo, exploded
FROM (SELECT *, split(Description, " ") as splitted FROM dfTable)
LATERAL VIEW explode(splitted) as exploded
```

This gives us the following result:

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

Maps

Maps are created by using the `map` function and key-value pairs of columns. You then can select them just like you might select from an array:

```
// in Scala
import org.apache.spark.sql.functions.map
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map"
```

```
# in Python
from pyspark.sql.functions import create_map
df.select(create_map(col("Description"), col("InvoiceNo")).alias("comp
  .show(2)
```

```
-- in SQL
SELECT map(Description, InvoiceNo) as complex_map FROM dfTable
WHERE Description IS NOT NULL
```

This produces the following result:

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

You can query them by using the proper key. A missing key returns `null`:

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

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

This gives us the following result:

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

You can also explode `map` types, which will turn them into columns:

```
// in Scala
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map"
  .selectExpr("explode(complex_map)").show(2)
```

```
# in Python
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map
  .selectExpr("explode(complex_map)").show(2)
```

This gives us the following result:

```
+--------------------+------+
|                 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 begin by creating a JSON column:

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

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

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

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

```
# in Python
from pyspark.sql.functions import get_json_object, json_tuple

jsonDF.select(
    get_json_object(col("jsonString"), "$.myJSONKey.myJSONValue[1]") a
    json_tuple(col("jsonString"), "myJSONKey")).show(2)
```

Here's the equivalent in SQL:

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

This results in the following table:

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

You can also turn a StructType into a JSON string by using the `to_json` function:

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

```
# in 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. You can use the `from_json` function to parse this (or other JSON data) back in. This naturally requires you to specify a schema, and optionally you can specify a map of options, as well:

```
// in 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")).show(
```

```
# in 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")).show
```

This gives us the following result:

```
+----------------------+--------------------+
|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 user-defined functions (UDFs) make it possible for you to write your own custom transformations using Python or Scala and even use external libraries. UDFs can take and return one or more columns as input. Spark UDFs are incredibly powerful because you can write them in several different programming languages; you do not need to create them in an esoteric format or domain-specific language. 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.

Although you can write UDFs 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 create a simple one for this example. Let's write a `power3` function that takes a number and raises it to a power of three:

```
// in Scala
val udfExampleDF = spark.range(5).toDF("num")
def power3(number:Double):Double = number * number * number
power3(2.0)
```

```
# in 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 "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 use 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.

When you use the function, there are essentially two different things that occur. If the function is written in Scala or Java, you can use it within the Java Virtual Machine (JVM). This means that there will be little performance penalty aside from the fact that you 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; we cover that in the section on optimization in Chapter 19.

If the function is written in Python, something quite different happens. Spark starts a Python process on the worker, serializes all of the data to a format that Python can understand (remember, it was in the JVM earlier), executes the function row by row on that data in the Python process, and then finally returns the results of the row operations to the JVM and Spark. Figure 6-2 provides an overview of the process.

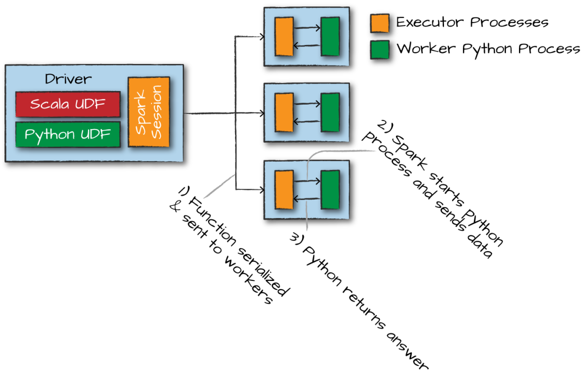

Figure 6-2. Figure caption

WARNING

Starting 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, after 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 or Java—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 you have an understanding of the process, let's work through an example. First, we need to register the function to make it available as a DataFrame function:

```
// in Scala
import org.apache.spark.sql.functions.udf
val power3udf = udf(power3(_:Double):Double)
```

We can use that just like any other DataFrame function:

```
// in Scala
udfExampleDF.select(power3udf(col("num"))).show()
```

The same applies to Python—first, we register it:

```
# in Python
from pyspark.sql.functions import udf
power3udf = udf(power3)
```

Then, we can use it in our DataFrame code:

```
# in Python
from pyspark.sql.functions import col
udfExampleDF.select(power3udf(col("num"))).show(2)
```

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

At this juncture, we can use this only as a 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 within SQL as well as across languages.

Let's register the function in Scala:

```
// in Scala
spark.udf.register("power3", power3(_:Double):Double)
udfExampleDF.selectExpr("power3(num)").show(2)
```

Because this function is registered with Spark SQL—and we've learned that any Spark SQL function or expression 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:

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

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

One thing we can also do to ensure 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, which 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 it is a best practice.

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

```
# in Python
from pyspark.sql.types import IntegerType, DoubleType
spark.udf.register("power3py", power3, DoubleType())
```

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

This is because the range 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 that 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, after we register them:

```
-- in SQL
```

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

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, you can also use 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 precompiled Scala and Java packages, so you'll need to specify them as a dependency:

```
-- in SQL
CREATE TEMPORARY FUNCTION myFunc AS 'com.organization.hive.udf.Functior
```

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

Conclusion

This chapter demonstrated how easy it is to extend Spark SQL to your own purposes and do so in a way that is not some esoteric, domain-specific language but rather simple functions that are easy to test and maintain without even using Spark! This is an amazingly powerful tool that you can use to specify sophisticated business logic that can run on five rows on your local machines or on terabytes of data on a 100-node cluster!