# player.select(to\_date(lit("2019-01-01")).alias("start"),to\_date(lit("2017-05-22")).alias("end")).select(months\_between(col("start"),col("end"))).show(1)Session II

How to work with different types of Data:

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

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://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset>

SQL Functions

<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$>

# 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.

from pyspark.sql.functions import col

df.where(col("InvoiceNo") != 536365).select("InvoiceNo", "Description") .show(5, False)

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.

from pyspark.sql.functions import instr

priceFilter = col("UnitPrice") > 600 d

escripFilter = instr(df.Description, "POSTAGE") >= 1 df.where(df.StockCode.isin("DOT")).where(priceFilter | descripFilter).show()

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

# 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 | descripFilter)).where("isExpensive").select("unitPrice", "isExpensive").show(5)

SQL Equivalent:

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))

# Working with Numbers

When working with big data, the second most common task you will do after filtering things is counting things.-counting, computation, finding correlations.

*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) + 5.*

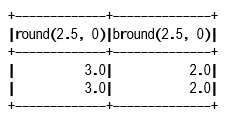
df.selectExpr( "CustomerId", "(POWER((Quantity \* UnitPrice), 2.0) + 5) as realQuantity").show(2)

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

# 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)

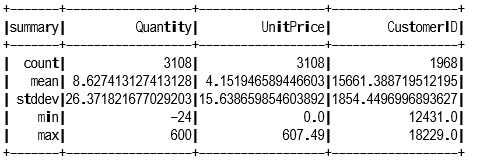


Another numerical task is to compute the correlation of two columns. For example, we can see the Pearson correlation coefficient for two columns to see if cheaper things are typically bought in greater quantities

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

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.

# in Python df.describe().show()



# Working with Strings

String manipulation shows up in nearly every data flow, and it’s worth explaining what you can do with strings.

The initcap function will capitalize every word in a given string when that word is separated from another by a space.

from pyspark.sql.functions import initcap

df.select(initcap(col("Description"))).show()

#player.select(initcap(lit("abc sh gchs vsd gsh"))).show()

+-----------------------------+

|initcap(abc sh gchs vsd gsh)|

+-----------------------------+

| Abc Sh Gchs Vsd Gsh|

+-----------------------------+

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

from pyspark.sql.functions import lower, upper

player.select(lit("Hello everyone"), lower(lit("Hello everyone")), upper(lit("Hello everyone"))).show(2)

+--------------+---------------------+---------------------+

|Hello everyone|lower(Hello everyone)|upper(Hello everyone)|

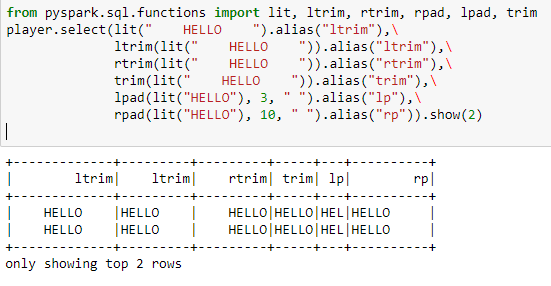
+--------------+---------------------+---------------------+

|Hello everyone| hello everyone| HELLO EVERYONE|

|Hello everyone| hello everyone| HELLO EVERYONE|

+--------------+---------------------+---------------------+

Removing space:



# Regular Expressions

Probably one of the most frequently performed tasks is s**earching 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

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:

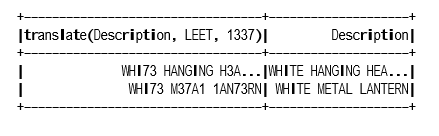
from pyspark.sql.functions import regexp\_replace

regex\_string = "BLACK|WHITE|RED|GREEN|BLUE"

df.select( regexp\_replace(col("Description"), regex\_string, "COLOR").alias("color\_clean"), col("Description")).show(2)

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

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 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)

# Working with Dates and Timestamps

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.

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.

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()

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 Python from pyspark.sql.functions import date\_add, date\_sub dateDF.select(date\_sub(col("today"), 5), date\_add(col("today"), 5)).show(1)

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:

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)

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.

**Spark will not throw an error if it cannot parse the date; rather, it will just return null**

**We will use two functions to fix this: to\_date and to\_timestamp. The former optionally expects a format, whereas the latter requires one:**

**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")**

Now let’s use an example of to\_timestamp, which always requires a format to be specified:

**from pyspark.sql.functions import to\_timestamp**

**cleanDateDF.select(to\_timestamp(col("date"), dateFormat)).show()**

# Working with Nulls

You should always use nulls to represent missing or empty data in your DataFrames.

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).

## 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 Python

from pyspark.sql.functions import coalesce

df.select(coalesce(col("Description"), col("CustomerId"))).show()

## **ifnull, nullIf, nvl, and nvl2**

* ifnull allows you to select the second value if the first is null, and defaults to the first.
* 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.
* nvl2 returns the second value if the first is not null; otherwise, it will return the last specified value

\*\*difference between ifnull and nvl?

## 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")

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")*

## 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")

For Integer:

df.na.fill(5:Integer),

For Doubles

df.na.fill(5:Double).

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:

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

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

# Working with Complex Types

## Structs

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 by using \*. This brings up all the columns to the top-level DataFrame:

complexDF.select("complex.\*")

## Arrays

### split

We do this by using the split function and specify the delimiter:

from pyspark.sql.functions import split

df.select(split(col("Description"), " ")).show(2)

We can query it like other columns:

df.select(split(col("Description"), " ").alias("array\_col") .selectExpr("array\_col[0]").show(2)

### **Array Length**

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

from pyspark.sql.functions import size

df.select(size(split(col("Description"), " "))).show(2)

# shows 5 and 3

### array\_contains

We can also see whether this array contains a value:

from pyspark.sql.functions import array\_contains

df.select(array\_contains(split(col("Description"), " "), "WHITE")).show(2)

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.

from pyspark.sql.functions import split, explode

df.withColumn("splitted", split(col("Description"), " ")).withColumn("exploded", explode(col("splitted"))) .select("Description", "InvoiceNo", "exploded").show(2)

Hello World

[“Hello”,”World”] (After Split)

After Explode:

“Hello”

“World”

## Map:

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.

from pyspark.sql.functions import create\_map

df.select(create\_map(col("Description"), col("InvoiceNo")).alias("complex\_map")).show(2)

To Select:

df.select(map(col("Description"), col("InvoiceNo")).alias("complex\_map")).selectExpr("complex\_map['WHITE METAL LANTERN']").show(2)

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

df.select(map(col("Description"), col("InvoiceNo")).alias("complex\_map"))\ .selectExpr("explode(complex\_map)").show(2)

# Working with JSON

You can operate directly on strings of JSON in Spark and parse from JSON or extract JSON objects.

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:

You can also turn a StructType into a JSON string by using the to\_json function:

from pyspark.sql.functions import to\_json

df.selectExpr("(InvoiceNo, Description) as myStruct").select(to\_json(col("myStruct")))

# User Defined Functions:

you can write UDFs in Scala, Python, or Java, there are performance considerations that you should be aware of.

udfExampleDF = spark.range(5).toDF("num")

def power3(double\_value): return double\_value \*\* 3 power3(2.0)

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.

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.

from pyspark.sql.functions import udf

power3udf = udf(power3)

Then, we can use it in our DataFrame code:

from pyspark.sql.functions import col

udfExampleDF.select(power3udf(col("num"))).show(2)

# Aggregations:

## Aggregation Functions

You can find most aggregation functions in the org.apache.spark.sql.functions package.

### Count

from pyspark.sql.functions import count

df.select(count("StockCode")).show()

### countDistinct

Sometimes, the total number is not relevant; rather, it’s the number of unique groups that you want.

from pyspark.sql.functions import countDistinct

df.select(countDistinct("StockCode")).show()

### approx\_count\_distinct

Often, we find ourselves working with large datasets and the exact distinct count is irrelevant. There are times when an approximation to a certain degree of accuracy will work just fine

from pyspark.sql.functions import approx\_count\_distinct df.select(approx\_count\_distinct("StockCode", 0.1)).show() \

You will notice that approx\_count\_distinct took another parameter with which you can specify the maximum estimation error allowed.

first and last

You can get the first and last values from a DataFrame by using these two obviously named functions.

from pyspark.sql.functions import first, last

df.select(first("StockCode"), last("StockCode")).show()

min and max

To extract the minimum and maximum values from a DataFrame, use the min and max functions:

from pyspark.sql.functions import min, max

df.select(min("Quantity"), max("Quantity")).show()

### sum

Another simple task is to add all the values in a row using the sum function:

from pyspark.sql.functions import sum

df.select(sum("Quantity")).show()

### sumDistinct

In addition to summing a total, you also can sum a distinct set of values by using the sumDistinct function:

from pyspark.sql.functions import sumDistinct

df.select(sumDistinct("Quantity")).show()

### avg

Although you can calculate average by dividing sum by count, Spark provides an easier way to get that value via the avg or mean functions.

from pyspark.sql.functions import sum, count, avg, expr

df.select( count("Quantity").alias("total\_transactions"), sum("Quantity").alias("total\_purchases"), avg("Quantity").alias("avg\_purchases"), expr("mean(Quantity)").alias("mean\_purchases"))\ .selectExpr( "total\_purchases/total\_transactions", "avg\_purchases", "mean\_purchases").show()

### Variance and Standard Deviation

The variance is the average of the squared differences from the mean, and the standard deviation is the square root of the variance

from pyspark.sql.functions import var\_pop, stddev\_pop

from pyspark.sql.functions import var\_samp, stddev\_samp

df.select(var\_pop("Quantity"), var\_samp("Quantity"), stddev\_pop("Quantity"), stddev\_samp("Quantity")).show()

### skewness and kurtosis

Skewness and kurtosis are both measurements of extreme points in your data. Skewness measures the asymmetry of the values in your data around the mean, whereas kurtosis is a measure of the tail of data.

from pyspark.sql.functions import skewness, kurtosis

df.select(skewness("Quantity"), kurtosis("Quantity")).show()

### Covariance and Correlation

We discussed single column aggregations, but some functions compare the interactions of the values in two difference columns together.

# in Python from pyspark.sql.functions import corr, covar\_pop, covar\_samp df.select(corr("InvoiceNo", "Quantity"), covar\_samp("InvoiceNo", "Quantity"), covar\_pop("InvoiceNo", "Quantity")).show()

### Aggregating to Complex Types

In Spark, you can perform aggregations not just of numerical values using formulas, you can also perform them on complex types. For example, we can collect a list of values present in a given column or only the unique values by collecting to a set.

from pyspark.sql.functions import collect\_set, collect\_list

df.agg(collect\_set("Country"), collect\_list("Country")).show()

## Grouping

df.groupBy("InvoiceNo", "CustomerId").count().show()

### Grouping with Expressions

Rather than passing that function as an expression into a select statement, we specify it as within agg. This makes it possible for you to pass-in arbitrary expressions that just need to have some aggregation specified.

from pyspark.sql.functions import count

df.groupBy("InvoiceNo").agg(count("Quantity").alias("quan"), expr("count(Quantity)")).show()

## Window Functions

A group-by takes data, and every row can go only into one grouping. A window function calculates a return value for every input row of a table based on a group of rows, called a frame. Each row can fall into one or more frames. A common use case is to take a look at a rolling average of some value for which each row represents one day. If you were to do this, each row would end up in seven different frames

**The first step to a window function is to create a window specification. Note that the partition by is unrelated to the partitioning scheme concept that we have covered thus far. It’s just a similar concept that describes how we will be breaking up our group. The ordering determines the ordering within a given partition, and, finally, the frame specification (the rowsBetween statement) states which rows will be included in the frame based on its reference to the current input row. In the following example, we look at all previous rows up to the current row:**

from pyspark.sql.window import Window

from pyspark.sql.functions import desc

windowSpec = Window\ .partitionBy("CustomerId", "date").orderBy(desc("Quantity")).rowsBetween(Window.unboundedPreceding, Window.currentRow)

from pyspark.sql.functions import max

maxPurchaseQuantity = max(col("Quantity")).over(windowSpec)

**Grouping Sets**

an aggregation across multiple groups. We achieve this by using grouping sets. Grouping sets are a low-level tool for combining sets of aggregations together. They give you the ability to create arbitrary aggregation in their group-by statements.

GROUP BY GROUPING SETS is equivalent to the UNION of two or more [GROUP BY](https://docs.snowflake.net/manuals/sql-reference/constructs/group-by.html) operations in the same result set:

* GROUP BY GROUPING SETS((a)) is equivalent to the single grouping set operation GROUP BY a.
* GROUP BY GROUPING SETS((a),(b)) is equivalent to GROUP BY a UNION ALL GROUP BY b.

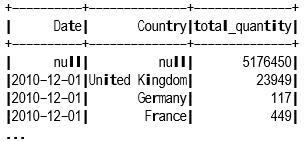
**The GROUPING SETS operator is only available in SQL. To perform the same in DataFrames, you use the rollup and cube operators—which allow us to get the same results. Let’s go through those.**

## Rolls Ups

A rollup is a **multidimensional aggregation** that performs a variety of group-by style calculations for us.

rolledUpDF = dfNoNull.rollup("Date", "Country").agg(sum("Quantity")).selectExpr("Date", "Country", "`sum(Quantity)` as total\_quantity").orderBy("Date")

rolledUpDF.show()



Now where you see the null values is where you’ll find the grand totals. A null in both rollup columns specifies the grand total across both of those columns:

### Cube

A cube takes the rollup to a level deeper. Rather than treating elements hierarchically, a cube does the same thing across all dimensions. This means that it won’t just go by date over the entire time period, but also the country.

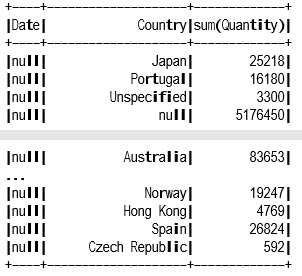
To pose this as a question again,

can you make a table that includes the following?

* The total across all dates and countries
* The total for each date across all countries
* The total for each country on each date
* The total for each country across all dates

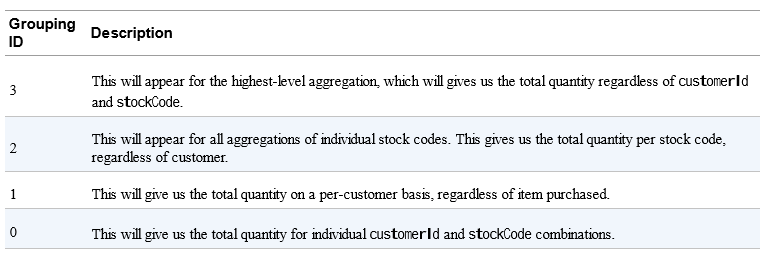
from pyspark.sql.functions import sum

dfNoNull.cube("Date", "Country").agg(sum(col("Quantity")))\ .select("Date", "Country", "sum(Quantity)").orderBy("Date").show()



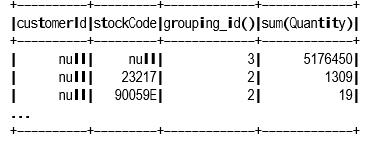
### Grouping Metadata

Sometimes when using cubes and rollups, you want to be able to query the aggregation levels so that you can easily filter them down accordingly. We can do this by using the grouping\_id, which gives us a column specifying the level of aggregation that we have in our result set.



// in Scala import org.apache.spark.sql.functions.{grouping\_id, sum, expr}

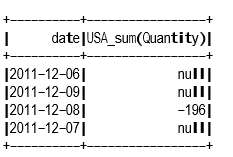
dfNoNull.cube("customerId", "stockCode").agg(grouping\_id(), sum("Quantity")) .orderBy(expr("grouping\_id()").desc) .show()



Pivot

Pivots make it possible for you to convert a row into a column. For example, in our current data we have a Country column. With a pivot, we can aggregate according to some function for each of those given countries and display them in an easy-to-query way:

pivoted = dfWithDate.groupBy("date").pivot("Country").sum()



This DataFrame will now have a column for every combination of country, numeric variable, and a column specifying the date. For example, for USA we have the following columns: USA\_sum(Quantity), USA\_sum(UnitPrice), USA\_sum(CustomerID). This represents one for each numeric column in our dataset (because we just performed an aggregation over all of them).

## User-Defined Aggregation Functions

UDAFs to compute custom calculations over groups of input data (as opposed to single rows). Spark maintains a single AggregationBuffer to store intermediate results for every group of input data.

To create a UDAF, you must inherit from the UserDefinedAggregateFunction base class and implement the following methods:

* inputSchema represents input arguments as a StructType
* bufferSchema represents intermediate UDAF results as a StructType
* dataType represents the return DataType deterministic is a Boolean value that specifies whether this UDAF will return the same result for a given input initialize allows you to initialize values of an aggregation buffer
* update describes how you should update the internal buffer based on a given row
* merge describes how two aggregation buffers should be merged
* evaluate will generate the final result of the aggregation

// in Scala

import org.apache.spark.sql.expressions.MutableAggregationBuffer

import org.apache.spark.sql.expressions.UserDefinedAggregateFunction

import org.apache.spark.sql.Row

import org.apache.spark.sql.types.\_

class BoolAnd extends UserDefinedAggregateFunction

{

def inputSchema: org.apache.spark.sql.types.StructType = StructType(StructField("value", BooleanType) :: Nil)

def bufferSchema: StructType = StructType( StructField("result", BooleanType) :: Nil )

def dataType: DataType = BooleanType

def deterministic: Boolean = true

def initialize(buffer: MutableAggregationBuffer): Unit = { buffer(0) = true }

def update(buffer: MutableAggregationBuffer, input: Row): Unit = { buffer(0) = buffer.getAs[Boolean](0) && input.getAs[Boolean](0) }

def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = { buffer1(0) = buffer1.getAs[Boolean](0) && buffer2.getAs[Boolean](0) }

def evaluate(buffer: Row): Any = { buffer(0) } }

Now, we simply instantiate our class and/or register it as a function:

// in Scala

val ba = new BoolAnd spark.udf.register("booland", ba)

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

spark.range(1) .selectExpr("explode(array(TRUE, TRUE, TRUE)) as t")

.selectExpr("explode(array(TRUE, FALSE, TRUE)) as f", "t") .select(ba(col("t")), expr("booland(f)")) .show()

**UDAFs are currently available only in Scala or Java**

# Join

A join brings together two sets of data, the left and the right, by comparing the value of one or more keys of the left and right and evaluating the result of a join expression that determines whether Spark should bring together the left set of data with the right set of data. The most common join expression, an equi-join, compares whether the specified keys in your left and right datasets are equal. If they are equal, Spark will combine the left and right datasets. The opposite is true for keys that do not match; Spark discards the rows that do not have matching keys.

## Join Types

Whereas the join expression determines whether two rows should join, the join type determines what should be in the result set.

There are a variety of different join types available in Spark for you to use:

* Inner joins (keep rows with keys that exist in the left and right datasets)
* Outer joins (keep rows with keys in either the left or right datasets)
* Left outer joins (keep rows with keys in the left dataset)
* Right outer joins (keep rows with keys in the right dataset)
* Left semi joins (keep the rows in the left, and only the left, dataset where the key appears in the right dataset)
* Left anti joins (keep the rows in the left, and only the left, dataset where they do not appear in the right dataset)
* Natural joins (perform a join by implicitly matching the columns between the two datasets
* with the same names)
* Cross (or Cartesian) joins (match every row in the left dataset with every row in the right dataset)

### Inner Joins

Inner joins evaluate the keys in both of the DataFrames or tables and include (and join together) only the rows that evaluate to true.

In the following example, we join the graduateProgram DataFrame with the person DataFrame to create a new DataFrame:

joinExpression = person["graduate\_program"] == graduateProgram['id']

Keys that do not exist in both DataFrames will not show in the resulting DataFrame. For example, the following expression would result in zero values in the resulting DataFrame:

**Inner joins are the default join**, so we just need to specify our left DataFrame and join the right in the JOIN expression:

person.join(graduateProgram, joinExpression).show()

**We can also specify this explicitly by passing in a third parameter, the joinType:**

joinType = "inner"

person.join(graduateProgram, joinExpression, joinType).show()

### Outer Joins

Outer joins evaluate the keys in both of the DataFrames or tables and includes (and joins together) the rows that evaluate to true or false. If there is no equivalent row in either the left or right DataFrame, Spark will insert null:

joinType = "outer"

person.join(graduateProgram, joinExpression, joinType).show()

### Left Outer Joins

Left outer joins evaluate the keys in both of the DataFrames or tables and includes all rows from the left DataFrame as well as any rows in the right DataFrame that have a match in the left DataFrame. If there is no equivalent row in the right DataFrame, Spark will insert null:

joinType = "left\_outer"

### Right Outer Joins

Right outer joins evaluate the keys in both of the DataFrames or tables and includes all rows from the right DataFrame as well as any rows in the left DataFrame that have a match in the right DataFrame. If there is no equivalent row in the left DataFrame, Spark will insert null:

joinType = "right\_outer"

### Left Semi Joins

Semi joins are a bit of a departure from the other joins. They do not actually include any values from the right DataFrame. They only compare values to see if the value exists in the second DataFrame. If the value does exist, those rows will be kept in the result, even if there are duplicate keys in the left DataFrame. Think of left semi joins as filters on a DataFrame, as opposed to the function of a conventional join:

joinType = "left\_semi"

### Left Anti Joins

Left anti joins are the opposite of left semi joins. Like left semi joins, they do not actually include any

values from the right DataFrame. They only compare values to see if the value exists in the second DataFrame. However, rather than keeping the values that exist in the second DataFrame, they keep only the values that do not have a corresponding key in the second DataFrame. Think of anti joins as a NOT IN SQL-style filter:

joinType = "left\_anti"

### Natural Joins

Natural joins make implicit guesses at the columns on which you would like to join. It finds matching columns and returns the results. Left, right, and outer natural joins are all supported.

### Cross (Cartesian) Joins

The last of our joins are cross-joins or cartesian products. Cross-joins in simplest terms are inner joins that do not specify a predicate. Cross joins will join every single row in the left DataFrame to ever single row in the right DataFrame. This will cause an absolute explosion in the number of rows contained in the resulting DataFrame. If you have 1,000 rows in each DataFrame, the cross-join of these will result in 1,000,000 (1,000 x 1,000) rows. For this reason, you must very explicitly state that you want a cross-join by using the cross join keyword:

joinType = "cross"

graduateProgram.join(person, joinExpression, joinType).show()

If you truly intend to have a cross-join, you can call that out explicitly:

person.crossJoin(graduateProgram).show()

## Challenges When Using Joins

### Joins on Complex Types

Even though this might seem like a challenge, it’s actually not. Any expression is a valid join expression, assuming that it returns a Boolean: om pyspark.sql.functions import expr

person.withColumnRenamed("id", "personId").join(sparkStatus, expr("array\_contains(spark\_status, id)")).show()

### Handling Duplicate Column Names

One of the tricky things that come up in joins is dealing with duplicate column names in your results DataFrame.

In a DataFrame, each column has a unique ID within Spark’s SQL Engine, Catalyst. This unique ID is purely internal and not something that you can directly reference. This makes it quite difficult to refer to a specific column when you have a DataFrame with duplicate column names.

Approach 1:

Different join expression When you have two keys that have the same name, probably the easiest fix is to change the join expression from a Boolean expression to a string or sequence. This automatically removes one of the columns for you during the join:

person.join(gradProgramDupe,"graduate\_program").select("graduate\_program").show()

Approach 2:

Dropping the column after the join Another approach is to drop the offending column after the join. When doing this, we need to refer to the column via the original source DataFrame. We can do this if the join uses the same key names or if the source DataFrames have columns that simply have the same name:

person.join(gradProgramDupe, joinExpr).drop(person.col("graduate\_program")) .select("graduate\_program").show()

Approach 3: Renaming a column before the join

val gradProgram3 = graduateProgram.withColumnRenamed("id", "grad\_id") val joinExpr = person.col("graduate\_program") === gradProgram3.col("grad\_id") person.join(gradProgram3, joinExpr).show()

# Data Sources

The ability to read and write from all different kinds of data sources and for the community to create its own contributions is arguably one of Spark’s greatest strengths.

Following are Spark’s core data sources:

* CSV
* JSON
* Parquet
* ORC
* JDBC/ODBC connections
* Plain-text files

Spark has numerous community-created data sources.

* Cassandra
* HBase
* MongoDB
* AWS Redshift XML And many, many others

# The Structure of the Data Sources API

## Read API Structure

The core structure for reading data is as follows:

DataFrameReader.format(...).option("key", "value").schema(...).load()

We will use this format to read from all of our data sources.

* Format is optional because by default Spark will use the Parquet format.
* option allows you to set key-value configurations to parameterize how you will read data.
* schema is optional if the data source provides a schema or if you intend to use schema inference..

Basics of Reading Data The foundation for reading data in Spark is the DataFrameReader. We access this through the SparkSession via the read attribute:

**spark.read**

spark.read.format("csv").option("mode", "FAILFAST") .option("inferSchema", "true").option("path", "path/to/file(s)").schema(someSchema).load()

There are a variety of ways in which you can set options; for example, you can build a map and pass in your configurations. For now, we’ll stick to the simple and explicit way that you just saw.

Read modes :

Reading data from an external source naturally entails encountering malformed data, especially when working with only semi-structured data sources. Read modes specify what will happen when Spark does come across malformed records.

|  |  |
| --- | --- |
| Permissive | Sets all fields to null when it encounters a corrupted record and places all corrupted records in a string column called \_corrupt\_record |
| dropMalformed | Drops the row that contains malformed records |
| failFast | Fails immediately upon encountering malformed records |

**The default is permissive.**

## Write API Structure

The core structure for writing data is as follows:

DataFrameWriter.**format**(...).**option**(...).partitionBy(...).bucketBy(...).sortBy( ...).**save**()

format is optional because by default, Spark will use the parquet format. option, again, allows us to configure how to write out our given data. PartitionBy, bucketBy, and sortBy work only for file-based data sources; you can use them to control the specific layout of files at the destination.

**dataFrame.write**

dataframe.write.format("csv").option("mode", "OVERWRITE").option("dateFormat", "yyyy-MM-dd").option("path", "path/to/file(s)").save()

Save modes Save modes specify what will happen if Spark finds data at the specified location (assuming all else equal).

|  |  |
| --- | --- |
| append | Appends the output files to the list of files that already exist at that location |
| overwrite | Will completely overwrite any data that already exists there |
| errorIfExists | Throws an error and fails the write if data or files already exist at the specified location |
| ignore | If data or files exist at the location, do nothing with the current DataFrame |

**The default is errorIfExists.**

# CSV (commma-separated values)

This is a common text file format in which each line represents a single record, and commas separate each field within a record. CSV files, while seeming well structured, are actually one of the trickiest file formats you will encounter because not many assumptions can be made in production scenarios about what they contain or how they are structured.

csvFile = spark.read.format("csv")\ .option("header", "true")\ .option("mode", "FAILFAST")\ .option("inferSchema", "true")\ .load("/data/flight-data/csv/2010-summary.csv")

**For instance, we can take our CSV file and write it out as a TSV file quite easily:**

csvFile.write.format("csv").mode("overwrite").option("sep", "\t")\ .save("/tmp/my-tsv-file.tsv")

When you list the destination directory, you can see that my-tsv-file is actually a folder with numerous files within it:

$ ls /tmp/my-tsv-file.tsv/

/tmp/my-tsv-file.tsv/part-00000-35cf9453-1943-4a8c-9c82-9f6ea9742b29.csv

This actually reflects the number of partitions in our DataFrame at the time we write it out. If we were to repartition our data before then, we would end up with a different number of files.

# JSON Files

In Spark, when we refer to JSON files, we refer to line delimited JSON files. This contrasts with files that have a large JSON object or array per file. The line-delimited versus multiline trade-off is controlled by a single option: multiLine. When you set this option to true, you can read an entire file as one json object and Spark will go through the work of parsing that into a DataFrame.

### Reading JSON Files

spark.read.format("json").option("mode", "FAILFAST").option("inferSchema", "true").load("/data/flight-data/json/2010-summary.json").show(5)

### Writing JSON Files

csvFile.write.format("json").mode("overwrite").save("/tmp/my-json-file.json")

# Parquet Files

Parquet is an open source column-oriented data store that provides a variety of storage optimizations, especially for analytics workloads. It provides **columnar compression**, which saves storage space and allows for reading individual columns instead of entire files.

It is a file format that works exceptionally well with Apache Spark and is in fact the default file format. We recommend writing data out to Parquet for long-term storage because reading from a Parquet file will always be more efficient than JSON or CSV. **Another advantage of Parquet is that it supports complex types.** This means that if your column is an array (which would fail with a CSV file, for example), map, or struct, you’ll still be able to read and write that file without issue. Here’s how to specify Parquet as the read format:

spark.read.format("parquet")

spark.read.format("parquet").load("/data/flight-data/parquet/2010-summary.parquet").show(5)

csvFile.write.format("parquet").mode("overwrite")\ .save("/tmp/my-parquet-file.parquet")

# ORC Files

ORC is a self-describing, type-aware columnar file format designed for Hadoop workloads. It is optimized for large streaming reads, but with integrated support for finding required rows quickly. ORC actually has no options for reading in data because Spark understands the file format quite well. An often-asked question is: What is the difference between ORC and Parquet? For the most part, they’re quite similar; the fundamental difference is that Parquet is further optimized for use with Spark, whereas ORC is further optimized for Hive.

spark.read.format("orc").load("/data/flight-data/orc/2010-summary.orc").show(5)

csvFile.write.format("orc").mode("overwrite").save("/tmp/my-json-file.orc")

# SQL Databases

Databases aren’t just a set of raw files, so there are more options to consider regarding how you connect to the database. Namely you’re going to need to begin considering things like **authentication** and **connectivity** (you’ll need to determine whether the network of your Spark cluster is connected to the network of your database system). To avoid the distraction of setting up a database for the purposes of this book, we provide a reference sample that runs on SQLite. We can skip a lot of these details by using SQLite, because it can work with minimal setup on your local machine with the limitation of not being able to work in a distributed setting. If you want to work through these examples in a distributed setting, you’ll want to connect to another kind of database.

To read and write from these databases, you need to do two things:

* include the Java Database Connectivity (JDBC) driver for you particular database on the spark classpath
* provide the proper JAR for the driver itself.

For example, to be able to read and write from PostgreSQL, you might run something like this:

./bin/spark-shell \ --driver-class-path postgresql-9.4.1207.jar \ --jars postgresql-9.4.1207.jar

## Reading from SQL Databases

When it comes to reading a file, SQL databases are no different from the other data sources that we looked at earlier. As with those sources, we specify the format and options, and then load in the data:

driver = "org.sqlite.JDBC"

path = "/data/flight-data/jdbc/my-sqlite.db"

url = "jdbc:sqlite:" + path

tablename = "flight\_info"

After you have defined the connection properties, you can test your connection to the database itself to ensure that it is functional. This is an excellent troubleshooting technique to confirm that your database is available to (at the very least) the Spark driver. This is much less relevant for SQLite because that is a file on your machine but if you were using something like MySQL, you could test the connection with the following:

import java.sql.DriverManager

val connection = DriverManager.getConnection(url)

connection.isClosed()

connection.close()

If this connection succeeds, you’re good to go. Let’s go ahead and read the DataFrame from the SQL table:

dbDataFrame = spark.read.format("jdbc").option("url", url).option("dbtable", tablename).option("driver", driver).load()

SQLite has rather simple configurations (no users, for example). Other databases, like PostgreSQL, require more configuration parameters. Let’s perform the same read that we just performed, except using PostgreSQL this time:

pgDF = spark.read.format("jdbc").option("driver", "org.postgresql.Driver").option("url", "jdbc:postgresql://database\_server").option("dbtable", "schema.tablename").option("user", "username").option("password", "my-secret-password").load()

As we create this DataFrame, it is no different from any other: you can query it, transform it, and join it without issue. You’ll also notice that there is already a schema, as well. That’s because Spark gathers this information from the table itself and maps the types to Spark data types. Let’s get only the distinct locations to verify that we can query it as expected:

dbDataFrame.select("DEST\_COUNTRY\_NAME").distinct().show(5)

Awesome, we can query the database! Before we proceed, there are a couple of nuanced details that are worth understanding