**Chapter 7. Aggregations**

Aggregating is the act of collecting something together and is a cornerstone of big data analytics. In an aggregation, you will specify a *key* or *grouping* and an *aggregation function* that specifies how you should transform one or more columns. This function must produce one result for each group, given multiple input values. Spark’s aggregation capabilities are sophisticated and mature, with a variety of different use cases and possibilities. In general, you use aggregations to summarize numerical data usually by means of some grouping. This might be a summation, a product, or simple counting. Also, with Spark you can aggregate any kind of value into an array, list, or map, as we will see in “Aggregating to Complex Types”.

In addition to working with any type of values, Spark also allows us to create the following groupings types:

* The simplest grouping is to just summarize a complete DataFrame by performing an aggregation in a select statement.
* A “group by” allows you to specify one or more keys as well as one or more aggregation functions to transform the value columns.
* A “window” gives you the ability to specify one or more keys as well as one or more aggregation functions to transform the value columns. However, the rows input to the function are somehow related to the current row.
* A “grouping set,” which you can use to aggregate at multiple different levels. Grouping sets are available as a primitive in SQL and via rollups and cubes in DataFrames.
* A “rollup” makes it possible for you to specify one or more keys as well as one or more aggregation functions to transform the value columns, which will be summarized hierarchically.
* A “cube” allows you to specify one or more keys as well as one or more aggregation functions to transform the value columns, which will be summarized across all combinations of columns.

Each grouping returns a RelationalGroupedDataset on which we specify our aggregations.

**NOTE**

An important thing to consider is how exact you need an answer to be. When performing calculations over big data, it can be quite expensive to get an *exact* answer to a question, and it’s often much cheaper to simply request an approximate to a reasonable degree of accuracy. You’ll note that we mention some approximation functions throughout the book and oftentimes this is a good opportunity to improve the speed and execution of your Spark jobs, especially for interactive and ad hoc analysis.

Let’s begin by reading in our data on purchases, repartitioning the data to have far fewer partitions (because we know it’s a small volume of data stored in a lot of small files), and caching the results for rapid access:

*// in Scala*

**val** df **=** spark.read.format("csv")

.option("header", "true")

.option("inferSchema", "true")

.load("/data/retail-data/all/\*.csv")

.coalesce(5)

df.cache()

df.createOrReplaceTempView("dfTable")

*# in Python*

df = spark.read.format("csv")\

.option("header", "true")\

.option("inferSchema", "true")\

.load("/data/retail-data/all/\*.csv")\

.coalesce(5)

df.cache()

df.createOrReplaceTempView("dfTable")

Here’s a sample of the data so that you can reference the output of some of the functions:

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

|InvoiceNo|StockCode| Description|Quantity| InvoiceDate|UnitPrice|Cu...

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

| 536365| 85123A|WHITE HANGING... | 6|12/1/2010 8:26| 2.55| ...

| 536365| 71053|WHITE METAL... | 6|12/1/2010 8:26| 3.39| ...

...

| 536367| 21755|LOVE BUILDING BLO...| 3|12/1/2010 8:34| 5.95| ...

| 536367| 21777|RECIPE BOX WITH M...| 4|12/1/2010 8:34| 7.95| ...

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

As mentioned, basic aggregations apply to an entire DataFrame. The simplest example is the count method:

df.count() == 541909

If you’ve been reading this book chapter by chapter, you know that count is actually an action as opposed to a transformation, and so it returns immediately. You can use count to get an idea of the total size of your dataset but another common pattern is to use it to cache an entire DataFrame in memory, just like we did in this example.

Now, this method is a bit of an outlier because it exists as a method (in this case) as opposed to a function and is eagerly evaluated instead of a lazy transformation. In the next section, we will see count used as a lazy function, as well.

**Aggregation Functions**

All aggregations are available as functions, in addition to the special cases that can appear on DataFrames or via .stat, like we saw in [Chapter 6](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch06.html#s2c3---working-with-different-types-of-data). You can find most aggregation functions in the [org.apache.spark.sql.functions package](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions%24).

**NOTE**

There are some gaps between the available SQL functions and the functions that we can import in Scala and Python. This changes every release, so it’s impossible to include a definitive list. This section covers the most common functions.

**count**

The first function worth going over is count, except in this example it will perform as a transformation instead of an action. In this case, we can do one of two things: specify a specific column to count, or all the columns by using count(\*) or count(1) to represent that we want to count every row as the literal one, as shown in this example:

*// in Scala*

**import** **org.apache.spark.sql.functions.count**

df.select(count("StockCode")).show() *// 541909*

*# in Python*

**from** **pyspark.sql.functions** **import** count

df.select(count("StockCode")).show() *# 541909*

*-- in SQL*

**SELECT** **COUNT**(\*) **FROM** dfTable

**WARNING**

There are a number of gotchas when it comes to null values and counting. For instance, when performing a count(\*), Spark will count null values (including rows containing all nulls). However, when counting an individual column, Spark will not count the null values.

**countDistinct**

Sometimes, the total number is not relevant; rather, it’s the number of unique groups that you want. To get this number, you can use the countDistinct function. This is a bit more relevant for individual columns:

*// in Scala*

**import** **org.apache.spark.sql.functions.countDistinct**

df.select(countDistinct("StockCode")).show() *// 4070*

*# in Python*

**from** **pyspark.sql.functions** **import** countDistinct

df.select(countDistinct("StockCode")).show() *# 4070*

*-- in SQL*

**SELECT** **COUNT**(**DISTINCT** \*) **FROM** DFTABLE

**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, and for that, you can use the approx\_count\_distinct function:

*// in Scala*

**import** **org.apache.spark.sql.functions.approx\_count\_distinct**

df.select(approx\_count\_distinct("StockCode", 0.1)).show() *// 3364*

*# in Python*

**from** **pyspark.sql.functions** **import** approx\_count\_distinct

df.select(approx\_count\_distinct("StockCode", 0.1)).show() *# 3364*

*-- in SQL*

**SELECT** approx\_count\_distinct(StockCode, 0.1) **FROM** DFTABLE

You will notice that approx\_count\_distinct took another parameter with which you can specify the maximum estimation error allowed. In this case, we specified a rather large error and thus receive an answer that is quite far off but does complete more quickly than countDistinct. You will see much greater performance gains with larger datasets.

**first and last**

You can get the first and last values from a DataFrame by using these two obviously named functions. This will be based on the rows in the DataFrame, not on the values in the DataFrame:

*// in Scala*

**import** **org.apache.spark.sql.functions.**{first, last}

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

*# in Python*

**from** **pyspark.sql.functions** **import** first, last

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

*-- in SQL*

**SELECT** **first**(StockCode), **last**(StockCode) **FROM** dfTable

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

|first(StockCode, false)|last(StockCode, false)|

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

| 85123A| 22138|

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

**min and max**

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

*// in Scala*

**import** **org.apache.spark.sql.functions.**{min, max}

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

*# in Python*

**from** **pyspark.sql.functions** **import** min, max

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

*-- in SQL*

**SELECT** **min**(Quantity), **max**(Quantity) **FROM** dfTable

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

|min(Quantity)|max(Quantity)|

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

| -80995| 80995|

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

**sum**

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

*// in Scala*

**import** **org.apache.spark.sql.functions.sum**

df.select(sum("Quantity")).show() *// 5176450*

*# in Python*

**from** **pyspark.sql.functions** **import** sum

df.select(sum("Quantity")).show() *# 5176450*

*-- in SQL*

**SELECT** **sum**(Quantity) **FROM** dfTable

**sumDistinct**

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

*// in Scala*

**import** **org.apache.spark.sql.functions.sumDistinct**

df.select(sumDistinct("Quantity")).show() *// 29310*

*# in Python*

**from** **pyspark.sql.functions** **import** sumDistinct

df.select(sumDistinct("Quantity")).show() *# 29310*

*-- in SQL*

**SELECT** **SUM**(Quantity) **FROM** dfTable *-- 29310*

**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. In this example, we use alias in order to more easily reuse these columns later:

*// in Scala*

**import** **org.apache.spark.sql.functions.**{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()

*# in Python*

**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()

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

|(total\_purchases / total\_transactions)| avg\_purchases| mean\_purchases|

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

| 9.55224954743324|9.55224954743324|9.55224954743324|

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

**NOTE**

You can also average all the distinct values by specifying distinct. In fact, most aggregate functions support doing so only on distinct values.

**Variance and Standard Deviation**

Calculating the mean naturally brings up questions about the variance and standard deviation. These are both measures of the spread of the data around the mean. The variance is the average of the squared differences from the mean, and the standard deviation is the square root of the variance. You can calculate these in Spark by using their respective functions. However, something to note is that Spark has both the formula for the sample standard deviation as well as the formula for the population standard deviation. These are fundamentally different statistical formulae, and we need to differentiate between them. By default, Spark performs the formula for the sample standard deviation or variance if you use the variance or stddev functions.

You can also specify these explicitly or refer to the population standard deviation or variance:

*// in Scala*

**import** **org.apache.spark.sql.functions.**{var\_pop, stddev\_pop}

**import** **org.apache.spark.sql.functions.**{var\_samp, stddev\_samp}

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

stddev\_pop("Quantity"), stddev\_samp("Quantity")).show()

*# in Python*

**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()

*-- in SQL*

**SELECT** var\_pop(Quantity), var\_samp(Quantity),

stddev\_pop(Quantity), stddev\_samp(Quantity)

**FROM** dfTable

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

| var\_pop(Quantity)|var\_samp(Quantity)|stddev\_pop(Quantity)|stddev\_samp(Quan...|

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

|47559.303646609056|47559.391409298754| 218.08095663447796| 218.081157850...|

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

**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. These are both relevant specifically when modeling your data as a probability distribution of a random variable. Although here we won’t go into the math behind these specifically, you can look up definitions quite easily on the internet. You can calculate these by using the functions:

**import** **org.apache.spark.sql.functions.**{skewness, kurtosis}

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

*# in Python*

**from** **pyspark.sql.functions** **import** skewness, kurtosis

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

*-- in SQL*

**SELECT** skewness(Quantity), kurtosis(Quantity) **FROM** dfTable

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

| skewness(Quantity)|kurtosis(Quantity)|

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

|-0.2640755761052562|119768.05495536952|

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

**Covariance and Correlation**

We discussed single column aggregations, but some functions compare the interactions of the values in two difference columns together. Two of these functions are cov and corr, for covariance and correlation, respectively. Correlation measures the Pearson correlation coefficient, which is scaled between –1 and +1. The covariance is scaled according to the inputs in the data.

Like the var function, covariance can be calculated either as the sample covariance or the population covariance. Therefore it can be important to specify which formula you want to use. Correlation has no notion of this and therefore does not have calculations for population or sample. Here’s how they work:

*// in Scala*

**import** **org.apache.spark.sql.functions.**{corr, covar\_pop, covar\_samp}

df.select(corr("InvoiceNo", "Quantity"), covar\_samp("InvoiceNo", "Quantity"),

covar\_pop("InvoiceNo", "Quantity")).show()

*# 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()

*-- in SQL*

**SELECT** corr(InvoiceNo, Quantity), covar\_samp(InvoiceNo, Quantity),

covar\_pop(InvoiceNo, Quantity)

**FROM** dfTable

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

|corr(InvoiceNo, Quantity)|covar\_samp(InvoiceNo, Quantity)|covar\_pop(InvoiceN...|

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

| 4.912186085635685E-4| 1052.7280543902734| 1052.7...|

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

**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.

You can use this to carry out some more programmatic access later on in the pipeline or pass the entire collection in a user-defined function (UDF):

*// in Scala*

**import** **org.apache.spark.sql.functions.**{collect\_set, collect\_list}

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

*# in Python*

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

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

*-- in SQL*

**SELECT** collect\_set(Country), collect\_set(Country) **FROM** dfTable

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

|collect\_set(Country)|collect\_list(Country)|

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

|[Portugal, Italy,...| [United Kingdom, ...|

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

**Grouping**

Thus far, we have performed only DataFrame-level aggregations. A more common task is to perform calculations based on *groups* in the data. This is typically done on categorical data for which we group our data on one column and perform some calculations on the other columns that end up in that group.

The best way to explain this is to begin performing some groupings. The first will be a count, just as we did before. We will group by each unique invoice number and get the count of items on that invoice. Note that this returns another DataFrame and is lazily performed.

We do this grouping in two phases. First we specify the column(s) on which we would like to group, and then we specify the aggregation(s). The first step returns a RelationalGroupedDataset, and the second step returns a DataFrame.

As mentioned, we can specify any number of columns on which we want to group:

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

*-- in SQL*

**SELECT** **count**(\*) **FROM** dfTable **GROUP** **BY** InvoiceNo, CustomerId

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

|InvoiceNo|CustomerId|count|

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

| 536846| 14573| 76|

...

| C544318| 12989| 1|

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

**Grouping with Expressions**

As we saw earlier, counting is a bit of a special case because it exists as a method. For this, usually we prefer to use the count function. 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. You can even do things like alias a column after transforming it for later use in your data flow:

*// in Scala*

**import** **org.apache.spark.sql.functions.count**

df.groupBy("InvoiceNo").agg(

count("Quantity").alias("quan"),

expr("count(Quantity)")).show()

*# in Python*

**from** **pyspark.sql.functions** **import** count

df.groupBy("InvoiceNo").agg(

count("Quantity").alias("quan"),

expr("count(Quantity)")).show()

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

|InvoiceNo|quan|count(Quantity)|

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

| 536596| 6| 6|

...

| C542604| 8| 8|

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

**Grouping with Maps**

Sometimes, it can be easier to specify your transformations as a series of Maps for which the key is the column, and the value is the aggregation function (as a string) that you would like to perform. You can reuse multiple column names if you specify them inline, as well:

*// in Scala*

df.groupBy("InvoiceNo").agg("Quantity"->"avg", "Quantity"->"stddev\_pop").show()

*# in Python*

df.groupBy("InvoiceNo").agg(expr("avg(Quantity)"),expr("stddev\_pop(Quantity)"))\

.show()

*-- in SQL*

**SELECT** **avg**(Quantity), stddev\_pop(Quantity), InvoiceNo **FROM** dfTable

**GROUP** **BY** InvoiceNo

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

|InvoiceNo| avg(Quantity)|stddev\_pop(Quantity)|

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

| 536596| 1.5| 1.1180339887498947|

...

| C542604| -8.0| 15.173990905493518|

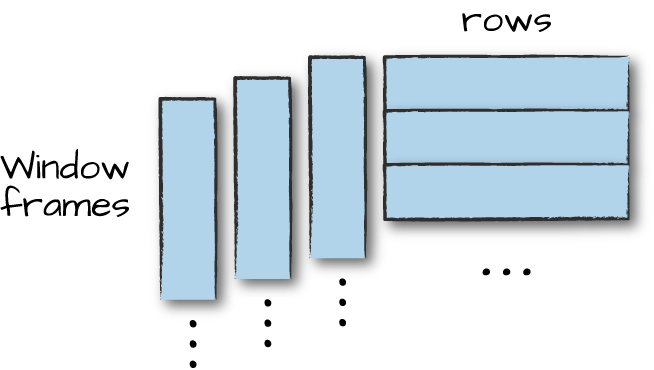
+---------+------------------+--------------------+

**Window Functions**

You can also use *window functions* to carry out some unique aggregations by either computing some aggregation on a specific “window” of data, which you define by using a reference to the current data. This window specification determines which rows will be passed in to this function. Now this is a bit abstract and probably similar to a standard group-by, so let’s differentiate them a bit more.

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. We cover defining frames a little later, but for your reference, Spark supports three kinds of window functions: ranking functions, analytic functions, and aggregate functions.

Figure 7-1 illustrates how a given row can fall into multiple frames.



*Figure 7-1. Visualizing window functions*

To demonstrate, we will add a date column that will convert our invoice date into a column that contains only date information (not time information, too):

*// in Scala*

**import** **org.apache.spark.sql.functions.**{col, to\_date}

**val** dfWithDate **=** df.withColumn("date", to\_date(col("InvoiceDate"),

"MM/d/yyyy H:mm"))

dfWithDate.createOrReplaceTempView("dfWithDate")

*# in Python*

**from** **pyspark.sql.functions** **import** col, to\_date

dfWithDate = df.withColumn("date", to\_date(col("InvoiceDate"), "MM/d/yyyy H:mm"))

dfWithDate.createOrReplaceTempView("dfWithDate")

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:

*// in Scala*

**import** **org.apache.spark.sql.expressions.Window**

**import** **org.apache.spark.sql.functions.col**

**val** windowSpec **=** **Window**

.partitionBy("CustomerId", "date")

.orderBy(col("Quantity").desc)

.rowsBetween(**Window**.unboundedPreceding, **Window**.currentRow)

*# in Python*

**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)

Now we want to use an aggregation function to learn more about each specific customer. An example might be establishing the maximum purchase quantity over all time. To answer this, we use the same aggregation functions that we saw earlier by passing a column name or expression. In addition, we indicate the window specification that defines to which frames of data this function will apply:

**import** **org.apache.spark.sql.functions.max**

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

*# in Python*

**from** **pyspark.sql.functions** **import** max

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

You will notice that this returns a column (or expressions). We can now use this in a DataFrame select statement. Before doing so, though, we will create the purchase quantity rank. To do that we use the dense\_rank function to determine which date had the maximum purchase quantity for every customer. We use dense\_rank as opposed to rank to avoid gaps in the ranking sequence when there are tied values (or in our case, duplicate rows):

*// in Scala*

**import** **org.apache.spark.sql.functions.**{dense\_rank, rank}

**val** purchaseDenseRank **=** dense\_rank().over(windowSpec)

**val** purchaseRank **=** rank().over(windowSpec)

*# in Python*

**from** **pyspark.sql.functions** **import** dense\_rank, rank

purchaseDenseRank = dense\_rank().over(windowSpec)

purchaseRank = rank().over(windowSpec)

This also returns a column that we can use in select statements. Now we can perform a select to view the calculated window values:

*// in Scala*

**import** **org.apache.spark.sql.functions.col**

dfWithDate.where("CustomerId IS NOT NULL").orderBy("CustomerId")

.select(

col("CustomerId"),

col("date"),

col("Quantity"),

purchaseRank.alias("quantityRank"),

purchaseDenseRank.alias("quantityDenseRank"),

maxPurchaseQuantity.alias("maxPurchaseQuantity")).show()

*# in Python*

**from** **pyspark.sql.functions** **import** col

dfWithDate.where("CustomerId IS NOT NULL").orderBy("CustomerId")\

.select(

col("CustomerId"),

col("date"),

col("Quantity"),

purchaseRank.alias("quantityRank"),

purchaseDenseRank.alias("quantityDenseRank"),

maxPurchaseQuantity.alias("maxPurchaseQuantity")).show()

*-- in SQL*

**SELECT** CustomerId, date, Quantity,

rank(Quantity) OVER (PARTITION **BY** CustomerId, date

**ORDER** **BY** Quantity **DESC** NULLS **LAST**

**ROWS** **BETWEEN**

UNBOUNDED PRECEDING **AND**

**CURRENT** **ROW**) **as** rank,

dense\_rank(Quantity) OVER (PARTITION **BY** CustomerId, date

**ORDER** **BY** Quantity **DESC** NULLS **LAST**

**ROWS** **BETWEEN**

UNBOUNDED PRECEDING **AND**

**CURRENT** **ROW**) **as** dRank,

**max**(Quantity) OVER (PARTITION **BY** CustomerId, date

**ORDER** **BY** Quantity **DESC** NULLS **LAST**

**ROWS** **BETWEEN**

UNBOUNDED PRECEDING **AND**

**CURRENT** **ROW**) **as** maxPurchase

**FROM** dfWithDate **WHERE** CustomerId **IS** **NOT** **NULL** **ORDER** **BY** CustomerId

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

|CustomerId| date|Quantity|quantityRank|quantityDenseRank|maxP...Quantity|

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

| 12346|2011-01-18| 74215| 1| 1| 74215|

| 12346|2011-01-18| -74215| 2| 2| 74215|

| 12347|2010-12-07| 36| 1| 1| 36|

| 12347|2010-12-07| 30| 2| 2| 36|

...

| 12347|2010-12-07| 12| 4| 4| 36|

| 12347|2010-12-07| 6| 17| 5| 36|

| 12347|2010-12-07| 6| 17| 5| 36|

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

**Grouping Sets**

Thus far in this chapter, we’ve seen simple group-by expressions that we can use to aggregate on a set of columns with the values in those columns. However, sometimes we want something a bit more complete—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.

Let’s work through an example to gain a better understanding. Here, we would like to get the total quantity of all stock codes and customers. To do so, we’ll use the following SQL expression:

*// in Scala*

**val** dfNoNull **=** dfWithDate.drop()

dfNoNull.createOrReplaceTempView("dfNoNull")

*# in Python*

dfNoNull = dfWithDate.drop()

dfNoNull.createOrReplaceTempView("dfNoNull")

*-- in SQL*

**SELECT** CustomerId, stockCode, **sum**(Quantity) **FROM** dfNoNull

**GROUP** **BY** customerId, stockCode

**ORDER** **BY** CustomerId **DESC**, stockCode **DESC**

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

|CustomerId|stockCode|sum(Quantity)|

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

| 18287| 85173| 48|

| 18287| 85040A| 48|

| 18287| 85039B| 120|

...

| 18287| 23269| 36|

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

You can do the exact same thing by using a grouping set:

*-- in SQL*

**SELECT** CustomerId, stockCode, **sum**(Quantity) **FROM** dfNoNull

**GROUP** **BY** customerId, stockCode **GROUPING** **SETS**((customerId, stockCode))

**ORDER** **BY** CustomerId **DESC**, stockCode **DESC**

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

|CustomerId|stockCode|sum(Quantity)|

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

| 18287| 85173| 48|

| 18287| 85040A| 48|

| 18287| 85039B| 120|

...

| 18287| 23269| 36|

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

**WARNING**

Grouping sets depend on null values for aggregation levels. If you do not filter-out null values, you will get incorrect results. This applies to cubes, rollups, and grouping sets.

Simple enough, but what if you *also* want to include the total number of items, regardless of customer or stock code? With a conventional group-by statement, this would be impossible. But, it’s simple with grouping sets: we simply specify that we would like to aggregate at that level, as well, in our grouping set. This is, effectively, the union of several different groupings together:

*-- in SQL*

**SELECT** CustomerId, stockCode, **sum**(Quantity) **FROM** dfNoNull

**GROUP** **BY** customerId, stockCode **GROUPING** **SETS**((customerId, stockCode),())

**ORDER** **BY** CustomerId **DESC**, stockCode **DESC**

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

|customerId|stockCode|sum(Quantity)|

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

| 18287| 85173| 48|

| 18287| 85040A| 48|

| 18287| 85039B| 120|

...

| 18287| 23269| 36|

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

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.

**Rollups**

Thus far, we’ve been looking at explicit groupings. When we set our grouping keys of multiple columns, Spark looks at those as well as the actual combinations that are visible in the dataset. A rollup is a multidimensional aggregation that performs a variety of group-by style calculations for us.

Let’s create a rollup that looks across time (with our new Date column) and space (with the Country column) and creates a new DataFrame that includes the grand total over all dates, the grand total for each date in the DataFrame, and the subtotal for each country on each date in the DataFrame:

**val** rolledUpDF **=** dfNoNull.rollup("Date", "Country").agg(sum("Quantity"))

.selectExpr("Date", "Country", "`sum(Quantity)` as total\_quantity")

.orderBy("Date")

rolledUpDF.show()

*# in Python*

rolledUpDF = dfNoNull.rollup("Date", "Country").agg(sum("Quantity"))\

.selectExpr("Date", "Country", "`sum(Quantity)` as total\_quantity")\

.orderBy("Date")

rolledUpDF.show()

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

| Date| Country|total\_quantity|

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

| null| null| 5176450|

|2010-12-01|United Kingdom| 23949|

|2010-12-01| Germany| 117|

|2010-12-01| France| 449|

...

|2010-12-03| France| 239|

|2010-12-03| Italy| 164|

|2010-12-03| Belgium| 528|

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

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:

rolledUpDF.where("Country IS NULL").show()

rolledUpDF.where("Date IS NULL").show()

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

|Date|Country|total\_quantity|

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

|null| null| 5176450|

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

**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

The method call is quite similar, but instead of calling rollup, we call cube:

*// in Scala*

dfNoNull.cube("Date", "Country").agg(sum(col("Quantity")))

.select("Date", "Country", "sum(Quantity)").orderBy("Date").show()

*# in Python*

**from** **pyspark.sql.functions** **import** sum

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

.select("Date", "Country", "sum(Quantity)").orderBy("Date").show()

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

|Date| Country|sum(Quantity)|

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

|null| Japan| 25218|

|null| Portugal| 16180|

|null| Unspecified| 3300|

|null| null| 5176450|

|null| Australia| 83653|

...

|null| Norway| 19247|

|null| Hong Kong| 4769|

|null| Spain| 26824|

|null| Czech Republic| 592|

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

This is a quick and easily accessible summary of nearly all of the information in our table, and it’s a great way to create a quick summary table that others can use later on.

**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. The query in the example that follows returns four distinct grouping IDs:

*Table 7-1. Purpose of grouping IDs*

|  |  |
| --- | --- |
| **Grouping ID** | **Description** |
| 3 | This will appear for the highest-level aggregation, which will gives us the total quantity regardless of customerId and stockCode. |
| 2 | This will appear for all aggregations of individual stock codes. This gives us the total quantity per stock code, regardless of customer. |
| 1 | This will give us the total quantity on a per-customer basis, regardless of item purchased. |
| 0 | This will give us the total quantity for individual customerId and stockCode combinations. |

This is a bit abstract, so it’s well worth trying out to understand the behavior yourself:

*// 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()

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

|customerId|stockCode|grouping\_id()|sum(Quantity)|

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

| null| null| 3| 5176450|

| null| 23217| 2| 1309|

| null| 90059E| 2| 19|

...

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

**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:

*// in Scala*

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

*# in Python*

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).

Here’s an example query and result from this data:

pivoted.where("date > '2011-12-05'").select("date" ,"`USA\_sum(Quantity)`").show()

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

| date|USA\_sum(Quantity)|

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

|2011-12-06| null|

|2011-12-09| null|

|2011-12-08| -196|

|2011-12-07| null|

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

Now all of the columns can be calculated with single groupings, but the value of a pivot comes down to how you would like to explore the data. It can be useful, if you have low enough cardinality in a certain column to transform it into columns so that users can see the schema and immediately know what to query for.

**User-Defined Aggregation Functions**

User-defined aggregation functions (UDAFs) are a way for users to define their own aggregation functions based on custom formulae or business rules. You can use 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

The following example implements a BoolAnd, which will inform us whether all the rows (for a given column) are true; if they’re not, it will return false:

*// 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()

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

|booland(t)|booland(f)|

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

| true| false|

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

UDAFs are currently available only in Scala or Java. However, in Spark 2.3, you will also be able to call Scala or Java UDFs and UDAFs by registering the function just as we showed in the UDF section in [Chapter 6](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch06.html#s2c3---working-with-different-types-of-data). For more information, go to [SPARK-19439](https://issues.apache.org/jira/browse/SPARK-19439).

**Conclusion**

This chapter walked through the different types and kinds of aggregations that you can perform in Spark. You learned about simple grouping-to window functions as well as rollups and cubes. [Chapter 8](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch08.html#s2c5---joins) discusses how to perform joins to combine different data sources together.