Ch.7 - Aggregations

carry out some unique aggregations by computing some aggregation on a specific “window” of data, which you define by using a reference to the current data

Spark can aggregate any kind of value into an array,list, or map

- The simplest grouping is to just summarize a complete DataFrame by performing aggregation in a **select** statement.
- **Group By -** allows you to specify one or more keys as well as aggregation functions to transform the value columns.
- **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.
- **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.
- **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
- **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.

Consider how exact you need an answer to be - 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. Look for opportunities to improve the speed and execution of your Spark jobs with approximation functions, especially for interactive and ad hoc analysis

Count is actually an action, not a transformation, and so it returns immediately. 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.

WARNING - NULL VALUES:
When performing a count(*), Spark will count null values (including rows containing all nulls). But when counting an individual column, Spark will not count the null values.

Functions

- count
- countDistinct

- approx_count_distinct
- first, last
- sum
- sumDistinct
- avg
- var - variance
- stddev
- skewness
- kurtosis
- cov- covariance
- corr

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

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 By default, Spark performs the formula forthe sample standard deviation or variance if you use the *variance* or *stddev* functions

explicitly refer to the population or standard deviation or 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 - measures the asymmetry of the values in your data around the mean
Kurtosis - measures the tail of data

Measure the extremities of your data. Both relevant specifically when modeling your data as a *probability distribution* of a random variable.

covariance and correlation
Correlation measures the Pearson correlation coefficient, which is scaled between –1 and +1.
Covariance is scaled according to the inputs in the data -it is measure of the joint variability of two random variables. *Like the var function, covariance can be calculated either as the sample covariance or the population covariance*. Correlation has no notion of this and therefore does not have calculations for population orsample

Spark can perform aggregations, not just of numerical values using formulas, but on complex

types. 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 define **programmatic access** later on in the pipeline or pass the entire collection in a **user-defined function (UDF)**

```
from pyspark.sql.functions import collect_set, collect_list
df.agg(collect_set("Country"), collect_list("Country")).show()
```

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.

Below - Group by each unique invoice number and get the count of items on that invoice - *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 - returns a *RelationalGroupedDataset*
- Then we specify the aggregation(s) - returns a DataFrame.

Counting is a bit of a special case because it exists as a method. 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 to pass in arbitrary expressions that have an aggregation specified. You can even do things like alias a column after transforming it for later use in your data flow:

```
from pyspark.sql.functions import count


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


display(test)
```

Can 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.

```
df.groupBy("InvoiceNo").agg(expr("avg(Quantity)"),expr("stddev_pop(Quantity)"))
```

```
\
  .show()
```

Window Functions
Unique aggregations by computing some aggregation on a specific “window” of data, which you define by using a reference to the current data. Window specification determines which rows will be passed in to this function.

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. Each row would end up in seven different frames. 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.

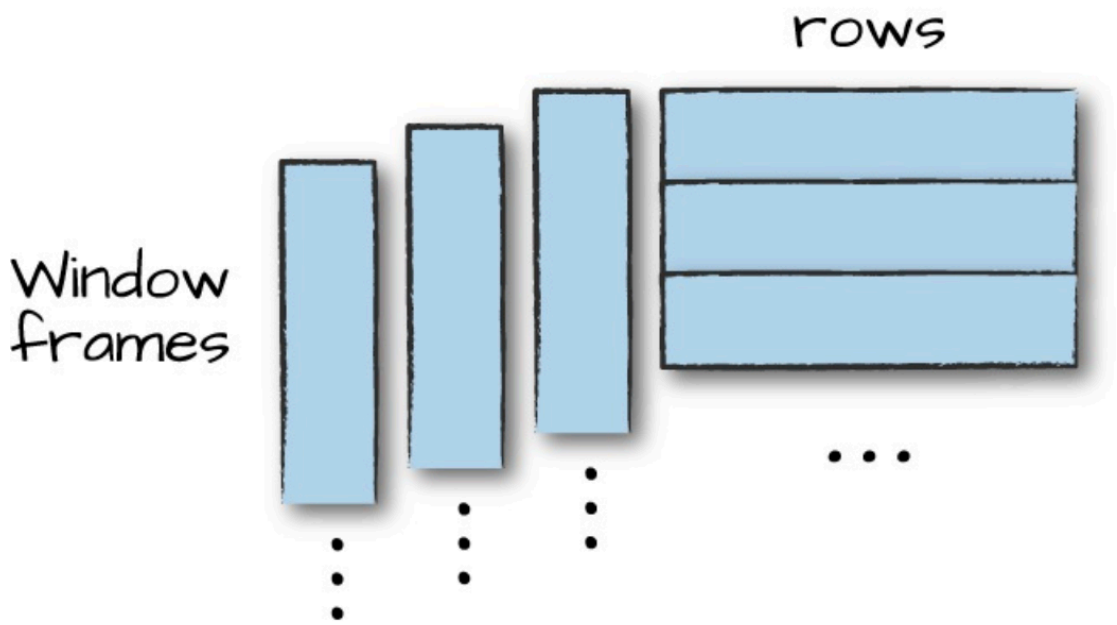

Figure 7-1. Visualizing window functions

The first step to a window function is to create a **window specification**. Note that the **partitionby** is unrelated to the partitioning scheme concept that we have covered thus far. 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 **currentinput** 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)
```

Use an aggregation function to learn more about each specific customer, establishing the maximum purchase quantity over all time. Use the same aggregation functions that we saw earlier, by passing a column name or expression, indicating the window specification that defines to which frames of data this function will apply:

```
from pyspark.sql.functions import max
maxPurchaseQuantity = max(col("Quantity")).over(windowSpec)
```

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*):

```
from pyspark.sql.functions import dense_rank, rank
purchaseDenseRank = dense_rank().over(windowSpec)
purchaseRank = rank().over(windowSpec)
```

Perform a select to view the calculated window values:

```
from pyspark.sql.functions import col

# EDIT: NOT IN TEXTBOOK - must set spark.sql.legacy.timeParserPolicy to LEGACY
spark.sql("set spark.sql.legacy.timeParserPolicy=LEGACY”)


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()
```

Grouping Sets
A bit more complete—an aggregation across multiple groups. 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.

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

The GROUPING SETS operator is only available in SQL. To perform the same in DataFrames, youuse the rollup and cube operators:

Rollup - a multidimensional aggregation that performs a variety of group-by style calculations for us.

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 theDataFrame

```
# rolledUpDF.where("Country IS NULL").show()
rolledUpDF.where("Date IS NULL && Country IS NULL").show()
```

Cube - 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. We can you make a tablethat 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 dateThe total for each country across all dates

Quick and easily accessible summary of nearly all of the information in our table, can create a summary table that others can use later on:

```
from pyspark.sql.functions import sum


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

Grouping Metadata

The **grouping_id** gives us a column specifying the level of aggregation that we have in our result set:

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.

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

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