Apache Spark SQL Day 2

Working with Different Types of Data

Lets look at working with the following data types:

- Booleans
- Numbers
- Strings

To begin, let's read in the DataFrame that we'll be using for this analysis:

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

df.printSchema()df.createOrReplaceTempView("dfTable")
```

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

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

```
// in Scala
import org.apache.spark.sql.functions.col

df.where(col("InvoiceNo")
.equalTo(536365))
.select("InvoiceNo", "Description")
.show(5, false)
```

You can also chain together Filters:

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


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

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

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


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

Working with Numbers

When working with big data, the second most common task you will do after filtering things is counting things

Imagine that we found out that we mis-recorded the quantity in our retail dataset and the true quantity is equal to (the current quantity * the unit price)^2 + 5. To implement this we will need a numerical function as well as the pow function that raises a column to the expressed power:

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


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

Pearson correlation coefficient

we can compute the Pearson correlation coefficient for two columns to see if cheaper things are typically bought in greater quantities. We can do this through a function as well as through the DataFrame statistic methods:

```
// in Scala
import org.apache.spark.sql.functions.{corr}

df.stat.corr("Quantity", "UnitPrice")
df.select(corr("Quantity", "UnitPrice")).show()
```

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

Compute summary statistics:

a common task is to compute summary statistics for a column or set of columns. We can use the describe method to achieve exactly this:

```
// in Scala
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.

What does this code do?

```
// in Scala
import org.apache.spark.sql.functions.{initcap}

df.select(initcap(col("Description"))).show(2, false)

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

Aggregations:

Aggregating is the act of collecting something together

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")
```

Count aggregation

```
df.count() == 541909
```

is count an action or transformation?

Aggregation Functions

count

```
// in Scala
import org.apache.spark.sql.functions.count
df.select(count("StockCode")).show() // 541909
```

what is the difference between count(*) and count("some_column_name") ?

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 SQL
SELECT COUNT(DISTINCT *) FROM DFTABLE
```

approx_count_distinct

```
// in Scala
import org.apache.spark.sql.functions.approx_count_distinct

df.select(approx_count_distinct("StockCode", 0.1)).show() // 3364
```

when would you use approx_count_distinct vs countDistinct?

Tasks for you

1. Write sql or scala code to compute the total of the quantity column.
2. Write sql or scala code to compute the total of the distinct quantity
3. Write sql or scala code to compute the average quantity value

Grouping

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).

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


-- in SQL

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