Spark Programming Tutorial (CSCI316)

September 7, 2019

Useful resources

https://spark.apache.org/docs/latest/quick-start.html
https://spark.apache.org/docs/latest/api/python/index.html

1 Basic Structure Operations

Basic Spark DataFrame concepts and operations

1.1 Create a Spark session

SparkSession is the entry of your application to a Spark cluster

```
In [1]: from pyspark.sql import SparkSession
```

```
In [2]: spark = SparkSession.builder.appName("CSCI316-week10") \
            .config("spark-master", "local") \
            .getOrCreate()
```

```
In [3]: spark
```

```
Out[3]: <pyspark.sql.session.SparkSession at 0x10d834b00>
```

1.1.1 Example 1: Flight data

```
In [4]: directory = "/Users/guoxinsu/Documents"
        df_FD = spark \
            .read \
            .format("json") \
            .load(directory+"/data/flight-data/json/2015-summary.json")
```

```
In [5]: df_FD.show(3)
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Romania|   15|
|    United States|            Croatia|    1|
```

```
|    United States|            Ireland|  344|
+-----------------+-------------------+-----+
only showing top 3 rows
```

```
In [6]: RDD = df_FD.rdd
        RDD.take(3)
```

```
Out[6]: [Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Romania', count=15),
         Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Croatia', count=1),
         Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Ireland', count=344)]
```

```
In [7]: RDD.map(lambda x: [x.DEST_COUNTRY_NAME, x.ORIGIN_COUNTRY_NAME]).take(3)
```

```
Out[7]: [['United States', 'Romania'],
         ['United States', 'Croatia'],
         ['United States', 'Ireland']]
```

DataFrames are *immutable*. - once a DataFrame is created, you cannot modify it - rather, your operations result in new DataFrames

A **schema** defines the column names and types of a DataFrame - Spark can implicitly infer the schema from the dataset

```
In [8]: df_FD.printSchema()
```

```
root
 |-- DEST_COUNTRY_NAME: string (nullable = true)
 |-- ORIGIN_COUNTRY_NAME: string (nullable = true)
 |-- count: long (nullable = true)
```

```
In [9]: df_FD.schema
```

```
Out[9]: StructType(List(StructField(DEST_COUNTRY_NAME,StringType,true),StructField(ORIGIN_COUN
```

- A schema is a StructType made up of a number of fields, StructFields, that have a name, a **Spark type**, a Boolean flag (whether that column can contain missing or null values), and users can optionally specify associated metadata with that column.

1.1.2 Spark Types

Data type	Value type in Python	API to access or create a data type
ByteType	int or long. Note: Numbers will be converted to 1-byte signed integer numbers at runtime. Ensure that numbers are within the range of –128 to 127.	ByteType()
ShortType	int or long. Note: Numbers will be converted to 2-byte signed integer numbers at runtime. Ensure that numbers are within the range of –32768 to 32767.	ShortType()
IntegerType	int or long. Note: Python has a lenient definition of "integer." Numbers that are too large will be rejected by Spark SQL if you use the IntegerType(). It's best practice to use LongType.	IntegerType()
LongType	long. Note: Numbers will be converted to 8-byte signed integer numbers at runtime. Ensure that numbers are within the range of –9223372036854775808 to 9223372036854775807. Otherwise, convert data to decimal.Decimal and use DecimalType.	LongType()
FloatType	float. Note: Numbers will be converted to 4-byte single-precision floating-point numbers at runtime.	FloatType()
DoubleType	float	DoubleType()
DecimalType	decimal.Decimal	DecimalType()
StringType	string	StringType()
BinaryType	bytearray	BinaryType()
BooleanType	bool	BooleanType()
TimestampType	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()
ArrayType	list, tuple, or array	ArrayType(elementType, [containsNull]). Note: The default value of containsNull is True.
MapType	dict	MapType(keyType, valueType, [valueContainsNull]). Note: The default value of valueContainsNull is True.
StructType	list or tuple	StructType(fields). Note: `fields` is a list of StructFields. Also, fields with the same name are not allowed.
StructField	The value type in Python of the data type of this field (for example, Int for a StructField with the data type IntegerType)	StructField(name, dataType, [nullable]) Note: The default value of nullable is True.

- Users can also create and enforces a specific schema on a DataFrame:

```
In [10]: from pyspark.sql.types import *
         myManualSchema = StructType([
             StructField("DEST_COUNTRY_NAME", StringType(), True),
             StructField("ORIGIN_COUNTRY_NAME", StringType(), True),
             StructField("count1", LongType(),
                           False, metadata={"hello":"world"})])
         df_FD1 = spark.read.format("json").schema(myManualSchema) \
             .load(directory+"/data/flight-data/json/2015-summary.json")
```

```
In [11]: df_FD1.printSchema()
```

```
root
 |-- DEST_COUNTRY_NAME: string (nullable = true)
```

```
 |-- ORIGIN_COUNTRY_NAME: string (nullable = true)
 |-- count1: long (nullable = true)
```

1.2 Load and Save DataFrames

```
In [12]: df_FD.limit(5).write.mode('overwrite').format("json").save("my_json")
```

```
In [13]: spark.read.format("json").load("my_json").show()
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Romania|   15|
|    United States|            Croatia|    1|
|    United States|            Ireland|  344|
|            Egypt|      United States|   15|
|    United States|              India|   62|
+-----------------+-------------------+-----+
```

File formats: json, parquet, jdbc, orc, libsvm, csv, text, etc.

1.3 Columns and Expressions

- Columns in Spark are logical constructions that simply represent a value computed on a per-record basis by means of an expression
  - Cannot manipulate an individual column outside the context of a DataFrame

```
In [14]: from pyspark.sql.functions import col
         col("someColumnName")
```

```
Out[14]: Column<b'someColumnName'>
```

```
In [15]: from pyspark.sql.functions import expr
         expr("(((someCol + 5) * 200) - 6) < otherCol")
         df_FD.select(col("DEST_COUNTRY_NAME")).show(4)
```

```
+-----------------+
|DEST_COUNTRY_NAME|
+-----------------+
|    United States|
|    United States|
|    United States|
|            Egypt|
+-----------------+
only showing top 4 rows
```

1.4 Records and Rows

- Each row in a DataFrame is a single record.
  - Spark represents this record as an object of type Row.
  - Spark manipulates Row objects using column expressions in order to produce usable values.

```
In [16]: from pyspark.sql import Row
         myRow = Row("Hello", None, 1, False)
         myRow[0], myRow[1], myRow[2], myRow[3]
```

```
Out[16]: ('Hello', None, 1, False)
```

1.5 DataFrame Operations

DataFrames support two types of operations: *transformations* and *actions*.

- A transformation on RDDs return a new RDD
- An action actually "computes an output" (e.g., count the number of records)

1.6 Common DataFrame Transformations

- add rows or columns
- remove rows or columns
- transform a row into a column (or vice versa)
- change the order of rows based on the values in columns
- ...

1.6.1 Select and SelectExpr Methods

- select and selectExpr allow you to do the DataFrame equivalent of SQL queries on a table of data

```
In [17]: df_FD.select("DEST_COUNTRY_NAME").show(2)
```

```
+-----------------+
|DEST_COUNTRY_NAME|
+-----------------+
|    United States|
|    United States|
+-----------------+
only showing top 2 rows
```

```
In [18]: df_FD.select("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME").show(2)
```

```
+----------------+-------------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|
+----------------+-------------------+
|    United States|            Romania|
|    United States|            Croatia|
+----------------+-------------------+
only showing top 2 rows
```

```
In [19]: df_FD.select(expr("DEST_COUNTRY_NAME"),
                   col("DEST_COUNTRY_NAME")).show(2)
```

```
+-----------------+-----------------+
|DEST_COUNTRY_NAME|DEST_COUNTRY_NAME|
+-----------------+-----------------+
|    United States|    United States|
|    United States|    United States|
+-----------------+-----------------+
only showing top 2 rows
```

```
In [20]: df_FD.select(expr("DEST_COUNTRY_NAME AS destination")) \
             .show(2)
```

```
+-------------+
|  destination|
+-------------+
|United States|
|United States|
+-------------+
only showing top 2 rows
```

Informally, "selectExpr" is "select" + "expr".

```
In [21]: df_FD.selectExpr("DEST_COUNTRY_NAME as newColumnName",
                       "DEST_COUNTRY_NAME").show(2)
```

```
+-------------+-----------------+
|newColumnName|DEST_COUNTRY_NAME|
+-------------+-----------------+
|United States|    United States|
|United States|    United States|
+-------------+-----------------+
only showing top 2 rows
```

```
In [22]: df_FD.selectExpr(
             "*", # all original columns
             "(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry") \
             .show(2)
```

```
+-----------------+-------------------+-----+-------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|withinCountry|
+-----------------+-------------------+-----+-------------+
|    United States|            Romania|   15|        false|
|    United States|            Croatia|    1|        false|
+-----------------+-------------------+-----+-------------+
only showing top 2 rows
```

```
In [23]: df_FD.selectExpr("avg(count)",
                   "count(distinct(DEST_COUNTRY_NAME))") \
             .show(2)
```

```
+-----------+---------------------------------+
| avg(count)|count(DISTINCT DEST_COUNTRY_NAME)|
+-----------+---------------------------------+
|1770.765625|                              132|
+-----------+---------------------------------+
```

1.6.2 Add columns

Although **select** or **selectExpr** can be used to add columns to a DataFrame, a more formal way is to use **withColumn**.

```
In [24]: df_FD.withColumn("withinCountry",
           expr("ORIGIN_COUNTRY_NAME == DEST_COUNTRY_NAME"))\
             .show(2)
```

```
+-----------------+-------------------+-----+-------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|withinCountry|
+-----------------+-------------------+-----+-------------+
|    United States|            Romania|   15|        false|
|    United States|            Croatia|    1|        false|
+-----------------+-------------------+-----+-------------+
only showing top 2 rows
```

1.6.3 Remove columns

```
In [25]: df_FD.drop("count").show(2)
```

```
+----------------+-------------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|
+----------------+-------------------+
|    United States|            Romania|
|    United States|            Croatia|
+----------------+-------------------+
only showing top 2 rows
```

1.6.4 Filter rows

- Create an *expression* on rows
  - which is either a string or built by a set of column operations
- Use it to filter out the rows that evaluate the expression to *true*

```
In [26]: df_FD.filter(col("count") < 2) \
             .where(col("ORIGIN_COUNTRY_NAME") != "Croatia")\
             .show(2)
```

```
+----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+----------------+-------------------+-----+
|    United States|          Singapore|    1|
|          Moldova|      United States|    1|
+----------------+-------------------+-----+
only showing top 2 rows
```

1.6.5 Get unique rows

```
In [27]: df_FD.select("ORIGIN_COUNTRY_NAME").distinct().count()
```

```
Out[27]: 125
```

```
In [28]: df_FD.select("ORIGIN_COUNTRY_NAME",
                      "DEST_COUNTRY_NAME") \
             .distinct().count()
```

```
Out[28]: 256
```

1.6.6 Random samples and splits

Random sampling and splitting are frequently used in machine learning * deal with large number of records * generate training and testing datasets

```
In [29]: seed = 5
         withReplacement = False
         fraction = 0.5
         df_FD.sample(withReplacement, fraction, seed).count()
```

```
Out[29]: 126
```

```
In [30]: df1_FD, df2_FD = df_FD.randomSplit([0.25, 0.75], seed)
         [df1_FD.count(), df2_FD.count()]
```

```
Out[30]: [60, 196]
```

1.6.7 Concatenate and append rows

- To append to a DataFrame, you *union* the original DataFrame along with the new one
- The two DataFrames *must* have the same schema and number of columns

```
In [33]: from pyspark.sql import Row
         schema = df_FD.schema
         newRows = [
           Row("New Country", "Other Country", 5),
           Row("New Country 2", "Other Country 3", 1)]
         parallelizedRows = spark.sparkContext.parallelize(newRows)
           # create an RDD of Rows (see below)
         newDF = spark.createDataFrame(parallelizedRows, schema)
```

```
In [34]: newDF.union(df_FD).show(3)
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|      New Country|      Other Country|    5|
|    New Country 2|    Other Country 3|    1|
|    United States|            Romania|   15|
+-----------------+-------------------+-----+
only showing top 3 rows
```

1.6.8 Sort rows

Use **sort** or **orderBy**, you can sort by one or multiple columns.

```
In [35]: df_FD.sort("count").show(3)
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|          Moldova|      United States|    1|
|    United States|          Singapore|    1|
```

```
|    United States|            Croatia|    1|
+-----------------+-------------------+-----+
only showing top 3 rows
```

```
In [36]: df_FD.orderBy(col("count"), col("DEST_COUNTRY_NAME")).show(3)
         # or df_FD.orderBy("count", "DEST_COUNTRY_NAME").show(3)
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|     Burkina Faso|      United States|    1|
|    Cote d'Ivoire|      United States|    1|
|           Cyprus|      United States|    1|
+-----------------+-------------------+-----+
only showing top 3 rows
```

- Specify the sorting orders

```
In [37]: from pyspark.sql.functions import desc, asc
         df_FD.orderBy(col("count").desc(),
                       col("DEST_COUNTRY_NAME").asc()
                      ).show(3)
```

```
+-----------------+-------------------+------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME| count|
+-----------------+-------------------+------+
|    United States|      United States|370002|
|    United States|             Canada|  8483|
|           Canada|      United States|  8399|
+-----------------+-------------------+------+
only showing top 3 rows
```

1.6.9 Collect DataFrames to the Driver

- Show( ) prints a DataFrame object in a table form
- Collect( ) retrieves all Row objects in a DataFrame
  - *If the dataset is large, calling the above two methods can be very expensive!*
- Take(n) retrieves n Row objects in a DataFrame

```
In [38]: df_FD.take(2)
```

```
Out[38]: [Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Romania', count=15),
          Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Croatia', count=1)]
```

```
In [39]: len(df_FD.collect())
```

```
Out[39]: 256
```

1.6.10 Convert DataFrames into Pandas dataframes and Numpy arrays

Often, you need to turn a DataFrame into a *local* Python data structure and continues the computation.

Spark DataFrames provide a convenient function **.toPandas( )** that converts themselves into Pandas dataframes.

Calling .values variable of a Pandas dataframe returns a Numpy array.

```
In [40]: arr_FD = df_FD.toPandas().values
         print(type(arr_FD))
         print(arr_FD.dtype)
```

```
<class 'numpy.ndarray'>
object
```

```
In [41]: arr_FD[0,:]
```

```
Out[41]: array(['United States', 'Romania', 15], dtype=object)
```

The resulted Numpy array is of "object" type. You can convert it to a suitable type using its "astype" function. For example,

```
In [42]: subarr_FD = arr_FD[:,2].astype(float)
         subarr_FD.dtype
```

```
Out[42]: dtype('float64')
```

2 Work with Different Data Types

- Booleans
- Numbers
- Strings
- Handling Null/None
- Complex types
- User-defined functions

2.0.1 Example 2: Retail data

```
In [43]: df_RD = spark.read.format("csv")\
           .option("header", "true")\
           .option("inferSchema", "true")\
           .load(directory+"/data/retail-data/by-day/2010-12-01.csv")
         df_RD.printSchema()
```

```
root
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: integer (nullable = true)
 |-- InvoiceDate: timestamp (nullable = true)
 |-- UnitPrice: double (nullable = true)
 |-- CustomerID: double (nullable = true)
 |-- Country: string (nullable = true)
```

```
In [44]: df_RD.show(2)
```

```
+---------+---------+--------------------+--------+-------------------+---------+----------+---
|InvoiceNo|StockCode|         Description|Quantity|        InvoiceDate|UnitPrice|CustomerID|
+---------+---------+--------------------+--------+-------------------+---------+----------+---
|   536365|   85123A|WHITE HANGING HEA...|       6|2010-12-01 08:26:00|     2.55|   17850.0|Un
|   536365|    71053| WHITE METAL LANTERN|       6|2010-12-01 08:26:00|     3.39|   17850.0|Un
+---------+---------+--------------------+--------+-------------------+---------+----------+---
only showing top 2 rows
```

2.0.2 Booleans

```
In [45]: from pyspark.sql.functions import col
         df_RD.where(col("InvoiceNo") != 536365)\
           .select("InvoiceNo", "Description")\
           .show(5, False)
```

```
+---------+-----------------------------+
|InvoiceNo|Description                  |
+---------+-----------------------------+
|536366   |HAND WARMER UNION JACK       |
|536366   |HAND WARMER RED POLKA DOT    |
|536367   |ASSORTED COLOUR BIRD ORNAMENT|
|536367   |POPPY'S PLAYHOUSE BEDROOM    |
|536367   |POPPY'S PLAYHOUSE KITCHEN    |
+---------+-----------------------------+
only showing top 5 rows
```

Boolean expressions: and, or

- You can specify Boolean expressions serially:

```
In [46]: from pyspark.sql.functions import instr
         # instr() takes a column and a string as argument
         # and returns a column that contains the count of
         # the string in the original column.
         priceFilter = col("UnitPrice") > 600 # column object
         descripFilter = instr(df_RD.Description,
                               "POSTAGE") >= 1 # column object
         df_RD.where(df_RD.StockCode.isin("DOT"))\
             .where(priceFilter | descripFilter)\
             .select("InvoiceNo", "StockCode",
                     "Description", "UnitPrice") \
             .show()
         # isin() returns a boolean expression that is
         # evaluated to true if the value of this expression
         # is contained by the evaluated values of the arguments.
```

```
+---------+---------+--------------+---------+
|InvoiceNo|StockCode|   Description|UnitPrice|
+---------+---------+--------------+---------+
|   536544|      DOT|DOTCOM POSTAGE|   569.77|
|   536592|      DOT|DOTCOM POSTAGE|   607.49|
+---------+---------+--------------+---------+
```

- Or you can specify a Boolean column:

```
In [47]: DOTCodeFilter = col("StockCode") == "DOT"
         priceFilter = col("UnitPrice") > 600
         descripFilter = instr(col("Description"), "POSTAGE") >= 1
         df_RD.withColumn("isExpensive",
              DOTCodeFilter & (priceFilter | descripFilter))\
           .where("isExpensive")\
           .select("InvoiceNo", "StockCode",
                   "Description", "isExpensive").show(5)
```

```
+---------+---------+--------------+-----------+
|InvoiceNo|StockCode|   Description|isExpensive|
+---------+---------+--------------+-----------+
|   536544|      DOT|DOTCOM POSTAGE|       true|
|   536592|      DOT|DOTCOM POSTAGE|       true|
+---------+---------+--------------+-----------+
```

Expressing **negation**

```
In [48]: from pyspark.sql.functions import expr
         df_RD.withColumn("isNotCheap", expr("NOT UnitPrice <= 100"))\
```

```
        .withColumn("isNotExpensive", ~expr("UnitPrice >=700"))\
        .where(col("isNotCheap") & col("isNotExpensive") )\
        .select("Description", "UnitPrice").show()
```

```
+--------------------+---------+
|         Description|UnitPrice|
+--------------------+---------+
|RUSTIC  SEVENTEEN...|    165.0|
|      DOTCOM POSTAGE|   569.77|
|      DOTCOM POSTAGE|   607.49|
+--------------------+---------+
```

2.0.3 Numbers and Column Arithmetic

```
In [49]: from pyspark.sql.functions import expr, pow
         realQuantity = pow(col("Quantity")
                            * col("UnitPrice"), 2) + 5
         df_RD.select(expr("CustomerId"),
                   realQuantity.alias("realQuantity"))\
             .show(2)
```

```
+----------+------------------+
|CustomerId|      realQuantity|
+----------+------------------+
|   17850.0|239.08999999999997|
|   17850.0|          418.7156|
+----------+------------------+
only showing top 2 rows
```

One common task in data analytics is to compute basic statistics for a or multiple columns.

- **count**, **mean**, **standard deviation**, **min** and **max**

```
In [50]: df_RD.select("Quantity", "UnitPrice")\
             .describe().show()
```

```
+-------+------------------+------------------+
|summary|          Quantity|         UnitPrice|
+-------+------------------+------------------+
|  count|              3108|              3108|
|   mean| 8.627413127413128| 4.151946589446603|
| stddev|26.371821677029203|15.638659854603892|
|    min|               -24|               0.0|
|    max|               600|            607.49|
+-------+------------------+------------------+
```

- **Quantiles**

∗ Spark implements an efficient algorithm to approximately compute quantiles

```
In [51]: quantileProbs = [0.025, 0.25, 0.5, 0.75, 0.975]
         relError = 0.05 # may be expensive if set to 0
         df_RD.stat.approxQuantile("UnitPrice",
                                   quantileProbs, relError)
```

```
Out[51]: [0.0, 1.65, 2.51, 4.21, 607.49]
```

2.0.4 Strings

Strings manipulation tasks are highly common in data analytics.

- regular expression extraction/substitution
- checking for simple string existence
- capitalisation and de-capitablisation
- …

```
In [52]: from pyspark.sql.functions import lower, upper
         df_RD.select(col("Description"),
             lower(col("Description")),
             upper(lower(col("Description")))).show(2)
```

```
+--------------------+--------------------+------------------------+
|         Description|  lower(Description)|upper(lower(Description))|
+--------------------+--------------------+------------------------+
|WHITE HANGING HEA...|white hanging hea...|     WHITE HANGING HEA...|
| WHITE METAL LANTERN| white metal lantern|      WHITE METAL LANTERN|
+--------------------+--------------------+------------------------+
only showing top 2 rows
```

regexp_replace( ) replaces one regular expression with a specific string.

```
In [53]: from pyspark.sql.functions import regexp_replace
         regex_string = "BLACK|WHITE|RED|GREEN|BLUE"
         df_RD.select(
           regexp_replace(col("Description"),
                          regex_string, "COLOR") \
             .alias("color_clean"),
           col("Description")).show(2)
```

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

```
| COLOR METAL LANTERN| WHITE METAL LANTERN|
+--------------------+--------------------+
only showing top 2 rows
```

translate( ) replace a given *character* with another.

- avoiding the tedious process of building a regular expression

```
In [54]: from pyspark.sql.functions import translate
         df_RD.select(translate(col("Description"),
                                "LEET", "?!#$"),
                   col("Description")).show(2)
```

```
+----------------------------------+--------------------+
|translate(Description, LEET, ?!#$)|         Description|
+----------------------------------+--------------------+
|              WHI$! HANGING H!A...|WHITE HANGING HEA...|
|               WHI$! M!$A? ?AN$!RN| WHITE METAL LANTERN|
+----------------------------------+--------------------+
only showing top 2 rows
```

regexp_extract( ) extracts a specific occurence of a string.

```
In [55]: from pyspark.sql.functions import regexp_extract
         extract_str = "(BLACK|WHITE|RED|GREEN|BLUE)"
         df_RD.select(
              regexp_extract(col("Description"),
                     extract_str, 1).alias("color_clean"),
              col("Description")).show(2)
```

```
+-----------+--------------------+
|color_clean|         Description|
+-----------+--------------------+
|      WHITE|WHITE HANGING HEA...|
|      WHITE| WHITE METAL LANTERN|
+-----------+--------------------+
only showing top 2 rows
```

instr( ) checks whether a substring is contained in a string.

- returns the number of occurence

```
In [56]: from pyspark.sql.functions import instr
         containsBlack = instr(col("Description"), "BLACK") >= 1
         containsWhite = instr(col("Description"), "WHITE") >= 1
         df_RD.withColumn("hasSimpleColor",
                          containsBlack | containsWhite)\
           .where("hasSimpleColor")\
           .select("Description").show(3, False)
```

```
+----------------------------------+
|Description                       |
+----------------------------------+
|WHITE HANGING HEART T-LIGHT HOLDER|
|WHITE METAL LANTERN               |
|RED WOOLLY HOTTIE WHITE HEART.    |
+----------------------------------+
only showing top 3 rows
```

2.0.5 Null data

Null data often needs to be filtered out.

- use **isNull( )** and **isNotNull( )** functions

```
In [57]: df_RD.count()
```

```
Out[57]: 3108
```

```
In [58]: df_RD.where(col("Description").isNull()).count()
```

```
Out[58]: 10
```

```
In [59]: df_RD.where(col("Description").isNotNull()).count()
```

```
Out[59]: 3098
```

```
In [60]: df_RD.printSchema()
```

```
root
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: integer (nullable = true)
 |-- InvoiceDate: timestamp (nullable = true)
 |-- UnitPrice: double (nullable = true)
 |-- CustomerID: double (nullable = true)
 |-- Country: string (nullable = true)
```

Note. The “nullable = …” is a hard constraint but just a reflection of type information about the source data. In other words, if you want to avoid null values, always filter the rows (even when “nullable = false”).

2.1 Complex Types

Define and manipulate data of complex types in DataFrame columns

2.1.1 Lists (Scala Arrays)

split( ) to turn a long string into a list.

```
In [61]: from pyspark.sql.functions import split
         df1 = df_RD.select("Description")
         df2 = df1 \
             .withColumn("splitted", split(col("Description"), " "))
         df2.show(2,False)
```

```
+-----------------------------------+-----------------------------------------+
|Description                        |splitted                                 |
+-----------------------------------+-----------------------------------------+
|WHITE HANGING HEART T-LIGHT HOLDER|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]|
|WHITE METAL LANTERN                |[WHITE, METAL, LANTERN]                  |
+-----------------------------------+-----------------------------------------+
only showing top 2 rows
```

explode( ) takes a column that consists of lists and creates one row per value for each list.

```
In [62]: from pyspark.sql.functions import explode
         df3 = df2.limit(2) \
             .withColumn("exploded", explode(col("splitted")))
         df3.show()
```

```
+--------------------+--------------------+--------+
|         Description|            splitted|exploded|
+--------------------+--------------------+--------+
|WHITE HANGING HEA...|[WHITE, HANGING, ...|   WHITE|
|WHITE HANGING HEA...|[WHITE, HANGING, ...| HANGING|
|WHITE HANGING HEA...|[WHITE, HANGING, ...|   HEART|
|WHITE HANGING HEA...|[WHITE, HANGING, ...| T-LIGHT|
|WHITE HANGING HEA...|[WHITE, HANGING, ...|  HOLDER|
| WHITE METAL LANTERN|[WHITE, METAL, LA...|   WHITE|
| WHITE METAL LANTERN|[WHITE, METAL, LA...|   METAL|
| WHITE METAL LANTERN|[WHITE, METAL, LA...| LANTERN|
+--------------------+--------------------+--------+
```

2.2 User-Defined Functions

Taking one or more columns as input, *user-defined functions* (UDFs) make it possible for you to write your own custom transformations using Python.

- UDFs are registered as temporary functions to be used in that specific SparkSession
- Can use external library (should be aware of the performance considerations)

```
In [63]: udfExampleDF = spark.range(3).toDF("num")
         udfExampleDF.show()
```

```
+---+
|num|
+---+
|  0|
|  1|
|  2|
+---+
```

Define the actual function.

```
In [64]: def power3(double_value):
           return double_value ** 3

         power3(2.0)
```

```
Out[64]: 8.0
```

Register them with Spark and use it.

```
In [65]: from pyspark.sql.functions import udf
         power3udf = udf(power3)
         udfExampleDF.select(power3udf(col("num"))).show()
```

```
+-----------+
|power3(num)|
+-----------+
|          0|
|          1|
|          8|
+-----------+
```

3 Aggregations

Aggregating is the act of collecting something together and is a cornerstone of big data analytics.

- **Direct aggregations**
  - produce a result (e.g., actions)
  - common statistics: count, min/max, average, variance, covariance, correlation, etc.

- **Grouping**

```
In [66]: from pyspark.sql.functions import *
         df_RD.groupBy("InvoiceNo").agg(
                 count("Quantity"),
                 sum("Quantity"),
                 avg("Quantity")).show(4)
```

```
+---------+---------------+-------------+------------------+
|InvoiceNo|count(Quantity)|sum(Quantity)|     avg(Quantity)|
+---------+---------------+-------------+------------------+
|   536596|              6|            9|               1.5|
|   536597|             28|           71|2.5357142857142856|
|   536414|              1|           56|              56.0|
|   536550|              1|            1|               1.0|
+---------+---------------+-------------+------------------+
only showing top 4 rows
```

Variance and Standard Deviation Spark has both the formula for the sample variance (resp., standard deviation) as well as the formula for the population variance (resp., standard deviation). By default, Spark uses the sample variance and the sample standard deviation.

```
In [67]: from pyspark.sql.functions import var_pop, stddev_pop
         from pyspark.sql.functions import var_samp, stddev_samp
         df_RD.select(
             var_pop("Quantity"), var_samp("Quantity"),
             stddev_pop("Quantity"), stddev_samp("Quantity")
                     ).show()
```

```
+-----------------+------------------+--------------------+---------------------+
|var_pop(Quantity)|var_samp(Quantity)|stddev_pop(Quantity)|stddev_samp(Quantity)|
+-----------------+------------------+--------------------+---------------------+
|695.2492099104054| 695.4729785650273|   26.36757876465727 8|    26.371821677029203|
+-----------------+------------------+--------------------+---------------------+
```

Covariance and Correlation Like the variance function, covariance can be calculated either as the sample covariance or the population covariance. But correlation has no notion of this.

```
In [68]: from pyspark.sql.functions import corr, covar_pop, covar_samp
         df_RD.select(
             corr("InvoiceNo", "Quantity"),
             covar_samp("InvoiceNo", "Quantity"),
             covar_pop("InvoiceNo", "Quantity")
                  ).show()
```

```
+-------------------------+-------------------------------+------------------------------+
|corr(InvoiceNo, Quantity)|covar_samp(InvoiceNo, Quantity)|covar_pop(InvoiceNo, Quantity)|
+-------------------------+-------------------------------+------------------------------+
|     -0.12225395743668731|             -235.56327681311157|             -235.4868448608685|
+-------------------------+-------------------------------+------------------------------+
```

4 Resilient Distributed Datasets (RDDs)

An RDD represents an immutable, partitioned collection of records that can be operated on in parallel. - Unlike DataFrames, where each record is a structured row containing fields with a known schema, in RDDs the records are just Java, Scala, or Python objects of the programmer's choosing.

Compared with DataFrames, RDDs - are more flexible in passing user-defined functions - provide explicit support of key-value structures

4.0.1 Create RDDs

To create an RDD, you can use a SparkContext object, which is already created as the variable "sc" or can be created using as follows:

```
In [69]: sc = spark.sparkContext
```

```
In [70]: sc
```

```
Out[70]: <SparkContext master=local[*] appName=CSCI316-week10>
```

```
In [71]: SPARK_HOME = "/usr/local/Cellar/apache-spark/2.3.0/libexec"
         lines = sc.textFile(SPARK_HOME+"/python/README.md")
         lines.take(5)
```

```
Out[71]: ['# Apache Spark',
          '',
          'Spark is a fast and general cluster computing system for Big Data. It provides',
          'high-level APIs in Scala, Java, Python, and R, and an optimized engine that',
          'supports general computation graphs for data analysis. It also supports a']
```

4.0.2 Convert between DataFrames and RDDs

- From DataFrames to RDDs

```
In [72]: df_RD.rdd.take(1)
```

```
Out[72]: [Row(InvoiceNo='536365', StockCode='85123A', Description='WHITE HANGING HEART T-LIGHT
```

- From RDDs to DataFrames

```
In [73]: from pyspark.sql import Row
         schema = df_FD.schema
         newRecords = [
           Row("New Country", "Other Country", 5),
           Row("New Country 2", "Other Country 3", 1)]
         rowRDD = sc.parallelize(newRecords)
         newDF = spark.createDataFrame(rowRDD, schema)
         newDF.show()
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|      New Country|      Other Country|    5|
|    New Country 2|    Other Country 3|    1|
+-----------------+-------------------+-----+
```

4.1 RDD Operations

Like DataFrames, RDDs implement a range of methods (operations), which can be divided into two types: *transformations* and *actions*.

Many DataFrames operations have RDD counterparts: **count( )**, **distinct( )**, **union( )**, **groupBy( )**, **join( )**, **randomSplit( )**, etc.

4.1.1 Passing functions

Most RDD transformations and some of the RDD actions rely heavily on **passing functions**

- Programmatically, those transformations and actions take functions as arguments
- Physically, those functions are passed from the driver program to the, local or distributed, Spark cluster

There are several common ways of passing function

- lambda expressions (anonymous functions)
- locallly defined functions
- top-level functions in a module

4.1.2 Filter

```
In [74]: # how many lines contain the word 'Spark'
         sparkLines = lines.filter(lambda s: "Spark" in s)
         sparkLines.count()
```

```
Out[74]: 11
```

```
In [75]: def containSpark(s):
             return "Spark" in s
         sparkLines = lines.filter(containSpark)
         sparkLines.count()
```

```
Out[75]: 11
```

```
In [76]: # filter out empty strings
         nonemptyLines = lines.filter(lambda s: s!="")
         print("%i empty lines" %
               (lines.count() - nonemptyLines.count())
              )

12 empty lines
```

4.1.3 Map

You use **map( )** to apply a function processing an RDD, record by record, to generate another RDD.

```
In [77]: nonemptyLines.take(2)

Out[77]: ['# Apache Spark',
          'Spark is a fast and general cluster computing system for Big Data. It provides']
```

```
In [78]: import re
         splittedLines = nonemptyLines \
             .map(lambda x: re.findall("\w+", x))
         splittedLines.take(2)

Out[78]: [['Apache', 'Spark'],
          ['Spark',
           'is',
           'a',
           'fast',
           'and',
           'general',
           'cluster',
           'computing',
           'system',
           'for',
           'Big',
           'Data',
           'It',
           'provides']]
```

4.1.4 FlatMap

flatMap( ) flattens an RDD before applying a function to it (like **map( )**)

```
In [79]: flattenedLines = splittedLines \
             .flatMap(lambda w: w)
         flattenedLines.take(6)

Out[79]: ['Apache', 'Spark', 'Spark', 'is', 'a', 'fast']
```

4.1.5 Reduce

reduce( ) is an action which is frequently used to produce an analytics output.

```
In [80]: # number of words in splittedLines
         splittedLines.map(lambda x:len(x))\
             .reduce(lambda x,y: x+y)
```

```
Out[80]: 291
```

```
In [81]: splittedLines.map(lambda x: len(x)) \
             .reduce(lambda x,y: x if x>= y else y)
```

```
Out[81]: 71
```

4.2 Pair RDDs

Pair RDDs provide explict support of processing *key-value* pairs.

4.2.1 Create pair RDDs

```
In [82]: pairs = flattenedLines\
           .map(lambda x: (x.lower(),1))
         pairs.take(6)
```

```
Out[82]: [('apache', 1), ('spark', 1), ('spark', 1), ('is', 1), ('a', 1), ('fast', 1)]
```

4.2.2 ReduceByKey

```
In [83]: # find the freqencies of words
         pairs.reduceByKey(lambda x,y: x+y)\
           .take(4)
```

```
Out[83]: [('is', 4), ('provides', 1), ('high', 1), ('in', 2)]
```

4.2.3 Basic RDD transformations

https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations ### Baisc RDD actions https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions