No SQL Workshop
(Windows Platform using Python)

Lab Tutorial: Data Retrieval Using SPARK Queries

- *A self-paced, self-guided developed for student learning*

By:

Dr. Venkat Ramanathan

NUS-ISS

Singapore 119 615

rvenkat@nus.edu.sg

Total Pages: 22

Contents

WORKING WITH SPARK QUERIES

(Instructions for Spark Query Lab using Windows Platform – Python Language)

The goal of this tutorial is to train you on how to create data access classes using the Spark to retrieve data from a NoSQL data source. The data so retrieved may be consumed by an application or interface such as dashboard or other visualisation tools. This tutorial assumes no previous knowledge of the Spark Query or functional programming way of retrieving data. By the end of this tutorial, you'll understand how to use the Spark Frameworks to select, insert, update, and delete database records.

For the purpose of learning the various data operations, we will use a select few 'data tables' (or entity sets) drawn from a traditional Video Rental Store System use case scenario. In this workshop we provide three such entities as below:

- Customers *(these are members who borrow movies/videos from the store)*
- Movies *(aka videos, that provide details regarding each movie in store)*
- Producers *(these are movie production company that produced the above Movies)*

For learning purposes, we will propose to demonstrate that the Customer data are stored in CSV file, Movies data are stored in a SQL database table, and the Producer data is stored as JSON.

However, for practising query in this workshop, the source of data is of secondary focus as that will be dealt in greater detail in the *Ingestion Lesson.*

Throughout this self-paced tutorial, demonstrations with detailed examples are made using the Customer data. Student may walk through these examples in the Lab session. For further consolidation of the learning, a set of exercises are provided based on Movie Data which the student should complete to attain the required learning outcome.

Creating Data Sources for this workshop

Before writing any code for querying the data, we will first set up the above data.

1. Set up your workstation with the required software.

   You must have done this in an earlier workshop session, if not follow the instructions given in that workshop to download and install the various software below:

   a. Java SE Development Kit (JDK 8)

   b. Python IDE – we are using Pycharm community edition 2020 for this course.

   c. Windows Binaries for Hadoop (winutils).

   d. Apache Spark – we are using version 2.2 for this workshop.

   e. MySQL– we are using version 8.0

   Please note that other higher versions may work, but may have some issues pending testing.

2. Download the required data files from the GitHub given below and place them in the "Workspace" folder that you had created while installing Python. In my machine I have used D:\python\workspace.

   For Demo and Practice steps we provide the following files containing the data:

   i. Customer.CSV
   ii. Country.CSV
   iii. Movies.CSV
   iv. Producers.JSON (for practicing alternate data format i.e. JSON files)
   v. Movies.SQL (for practicing alternate data source i.e. MySQL RDBMS)
   vi. CustomerNoHdr.CSV

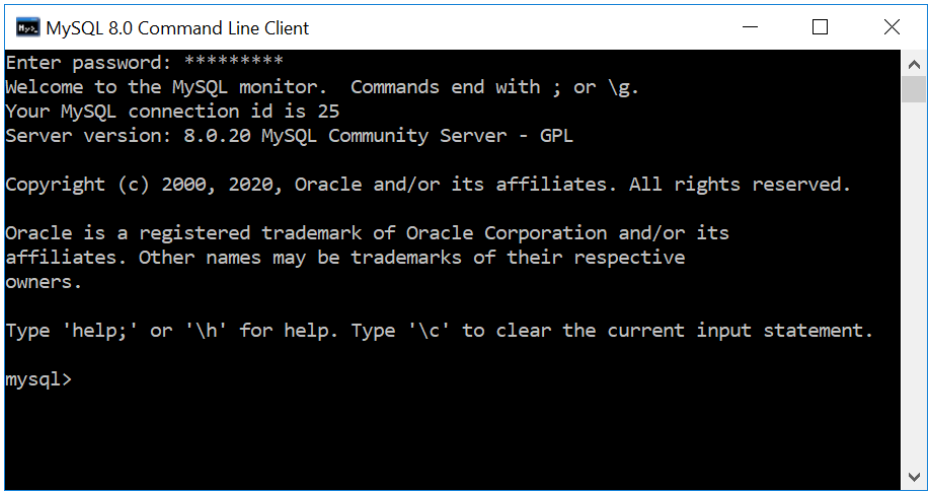
   The above files can be downloaded from: https://github.com/rvenkatiss/BEAD_DATA

   You can skip step 3(a-f) below and do it only when you wish to try MySQL. Suggest you try your workshops using CSV & JSON only to speed up completing your workshop… SKIP STEP 3 below and do it when you wish to try retrieving from MySQL.

3. For testing the MySQL source for the Movies data *(if you choose to attempt an SQL source),* you may use the following steps:

   a. Make sure you have installed the MySQL in your machine.

   b. Open the **MySQL Command Line Client** from the start menu (you may need to specify the root password that you created at the time of installation). The following will be the window:



   c. MySQL will be launched with a *mysql>* prompt. Issue the following command to create a new database called VideoShop (use this database for loading tables and practicing queries).

```
mysql>  create schema VideoShop ;
```

d. Copy the **Movies.SQL** file to an appropriate folder (this file contains the SQL statements to create Movie table and insert records commands for the movie data set). In my case I copied it to *D:\scala\workspace* folder.

e. From the MySQL command prompt issue the following commands *(specify the path that you have copied the file to).* These commands will execute and ensure that there are no errors.

```
mysql>  use VideoShop;
mysql>  source D:\scala\workspace\Movies.SQL ;
```

f. Use these commands to verify that the table has been created and data populated.

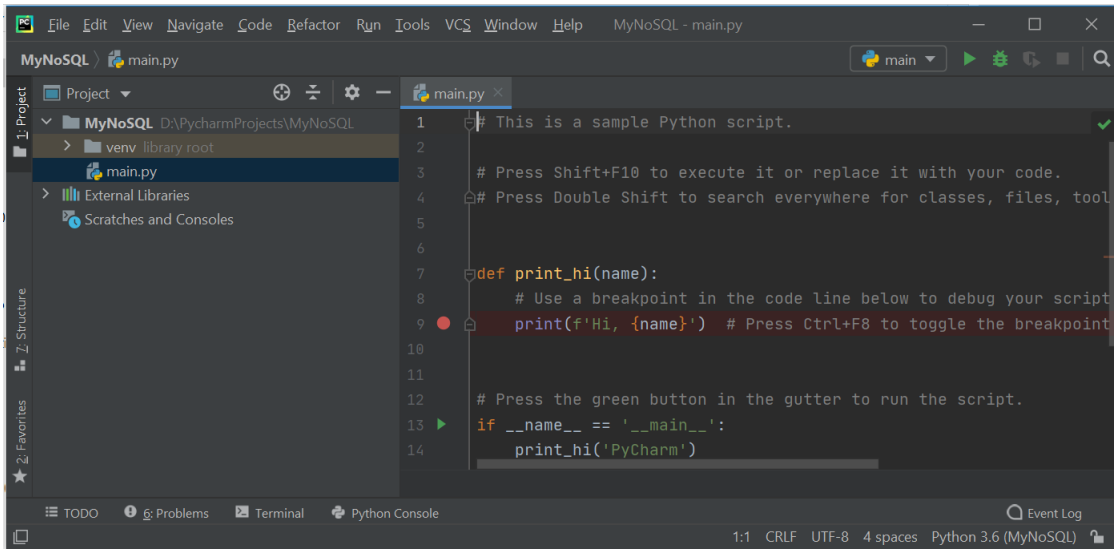
```
mysql>  show tables;
mysql>  select * from Movies ;
```

You have now set up the MySQL data set. The codes for retrieving data from Spark SQL will be discussed in a later section.

Note: Instead of using the command line interpreter described above, you can alternately use MySQL Workbench IDE. In this case you need to connect to the database and then using the query window execute the Movies.SQL file.

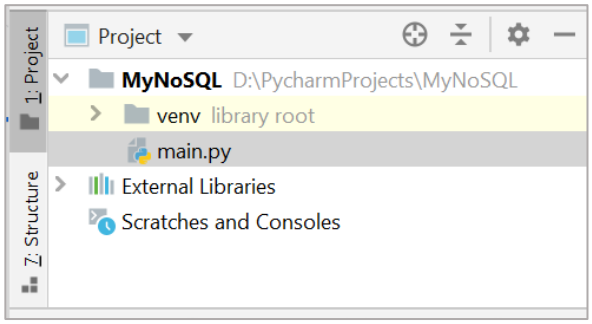
IDE for writing and testing Spark SQL

- To launch the PyCharm IDE, double click on the icon shown in right from desktop
- You will have the following screen:



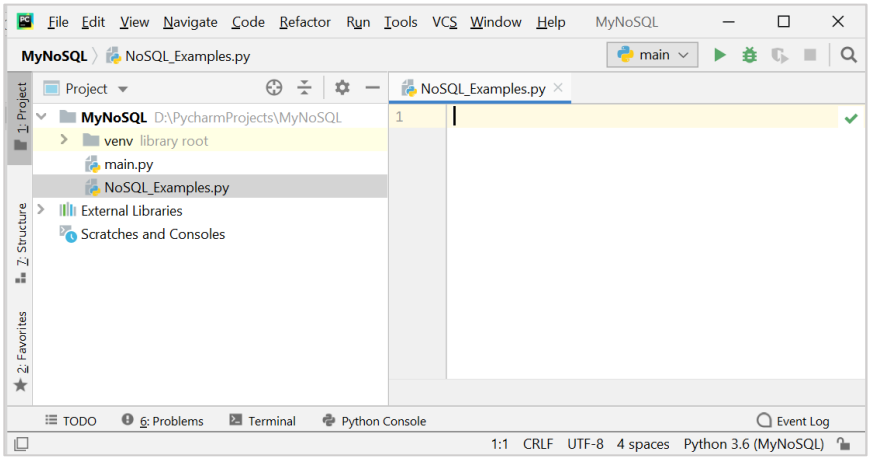
- You will be mostly using **the middle pane** to write your codes. You will also be using left pane (Project Explorer) for browsing project files and the bottom pane for viewing compile outcomes or outputs. *Note: If you do not see the Project Explorer, enable it using View-Tool Windows-Project.*

- Create a new Python Project; use the file menu for this. You will notice a project gets created once you complete the pop up prompt. I have created one called **MyNoSQL** and you will see the following on the project browser:



Note: A folder for the project will be created in the path you have specified, and all classes/programs that you create in subsequent steps will be placed in the this folder. In my case, this path is: **D:\PycharmProjects\MyNoSQL.**

- Create a new **Python Program** in the project; use the File menu for this or right click on the **MyNoSQL** project in the Package explorer. You may provide a suitable class name *(I have provided* **NoSQL_Examples** *for my examples).* Alternately you may use the main.py that was generated when you created the project. If you have created a new file, then you will find an empty file as below:



Since we wish to run this code as the start up for the project you will need to add the main function to this as shown below:

```
def main():
    print("Hello snake!")

if __name__ == '__main__':
    main()
```

- To test if all are good you may add a print message in the main method e.g.,

```
print ("Hello Snake")
```

- Include the following construct to invoke the main() function:

```
if __name__ =='__main__'
```

- Run the project to see that the program compiles and executes. You should see the *Hello Snake* message in the bottom panel.

Setting up Spark Libraries for the Project (MyNoSQL) created above

In order to write Spark Queries the respective libraries have to be installed for the project. *Please note this is for every project.*

To add PySpark:

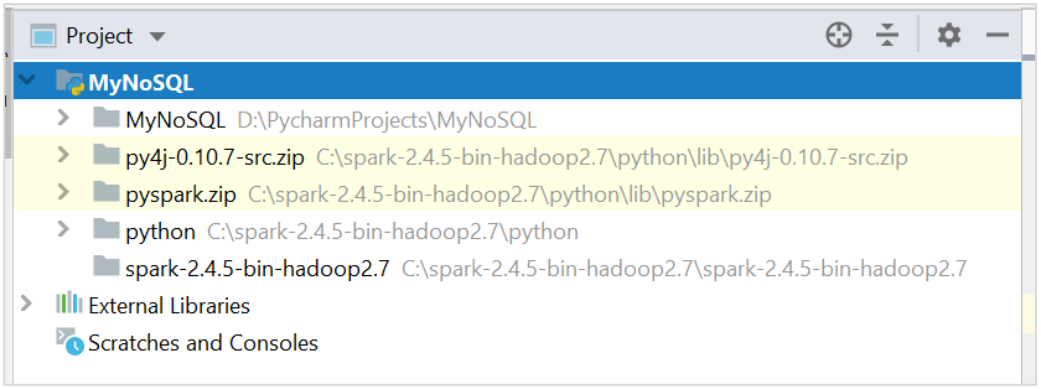
i. Choose menu **`File-Settings`**

ii. From the pop up window expand **`Project:MyNoSQL`** *(where MyNoSQL is your project name)*

iii. Choose **`Project Interpreter`** *(refer figure below for the pop up)*

iv. From the right pane choose **`the + symbol`**

v. Once the library window pops up, type pyspark. Select pyspark from the list and click on Install button. Once the installation is over, you will also notice the inclusions reflected in the Project Panel of the IDE as shown below:



Spark Query Examples

In this section we systematically walk through various queries starting from the simple ones and then build on to more complex queries. Walking through these self-guiding examples facilitates the students to gain comfort in handling various types of queries.

Python Spark provides the option of embedding standard SQL queries. While we introduce it later for completeness, we will confine to functional programming approach in this workshop which is the more acceptable way of coding in the modern day systems.

To get participants to speed, we confine to the CSV data deployed in a local folder, and walk the Scala Queries. It should be noted, that the queries themselves are IDENTICAL and INDEPENDENT of the data source and hence will work irrespective of whether the source is csv, json or rdbms database.

For completeness after demonstrating the queries, we will take up different data source types as well as storage such as HDFS clusters in the subsequent section.

Example 1: Retrieving all data from the Customers Dataset

Enter the following code in the **main()** function

```
from pyspark import SparkContext, SparkConf
from pyspark.sql import *
from pyspark.sql.functions import *
from pyspark.sql.types import *

def main():
    cnfg = SparkConf().setAppName("CustomerApplication").setMaster("local[2]")
    sc = SparkContext(conf=cnfg)
    spark = SparkSession(sc)

    inputFilePath = "D:\\workspace\\Customer.csv"

    df = ( spark.read
            .option("header", "true")
            .option("inferSchema", "true")
            .csv(inputFilePath)  )
    df.show()

if __name__ == '__main__':
    main()
```

- **SparkConf** loads the spark configurations from Scala.
- The variable **sc** provides the Spark data context.
- We will use the data frame (denoted by **df**) to perform queries in this and subsequent examples.
- The data frames **show** method outputs the data as below:

```
+----------+--------------------+--------------+---+------+-----------+--------------------+---------+-
|CustomerID|        CustomerName|MemberCategory|Age|Gender|AmountSpent|             Address|     City|O
+----------+--------------------+--------------+---+------+-----------+--------------------+---------+-
|      1000|        Lou Anna Tan|             A| 29|     F|       4.14|Blk 26, Telok Bla...|Frankfurt|
|      1001|      Wong Sook Huey|             A| 37|     F|       67.1|Blk 1007 Teresa V...|Singapore|
|      1002|       Ng Choon Seng|             C| 23|     M|      63.18|Blk 63 Bishan St ...|  Toronto|
|      1003|      Chew Teck Kuan|             C| 63|     M|      64.49|Blk 109 Bedok Nor...|Singapore|
|      1111|           Steven Ou|             B| 61|     M|      44.51|Blk 244, Bukit Pa...|     Rio |
|      1634|  Sridharan Jayanthi|             A| 55|     F|      61.51|Blk 232, Jurong E...|Singapore|
|      1681|         Terence Lim|             C| 30|     M|      59.62|Blk 99, Balestier...|  Columbo|
|      1810|         Vanessa Ong|             C| 32|     F|      80.97|Blk 20, Eunos Cre...|Singapore|
|      1811|       Koh Ting Ting|             B| 57|     F|      21.52|Blk 61, Upper Pay...|Singapore|
|      1818|    Chionh Choon Lee|             A| 57|     M|       7.13|Blk 89, Zion Road...|Singapore|
|      2131|                 Jon|             A| 64|     F|      83.45|Block 88 Demsey R...|   Zurich|
|      2233|       Too Siew Hong|             B| 35|     F|      21.83|Blk 749, Pasir Ri...|Singapore|
|      2270|   Chao Tah Jin Alex|             C| 22|     M|      87.07|Blk 12, Dover Clo...|Singapore|
|      2323|        Richard Kwan|             A| 26|     M|      89.52|Blk 27, Marine Cr...|Singapore|
|      2345| Ng Teck Kie Anthony|             A| 56|     M|      73.93|Blk 105, Gangsa R...|Singapore|
|      2626|          Steven Teo|             A| 56|     M|       8.17|Blk 253, Kim Keat...|Singapore|
|      2669|   Boh Lee Ming Lynn|             C| 23|     F|      41.99|Blk 671, Woodland...|Singapore|
|      2688|Kathleen Loh Swat...|             A| 38|     F|      39.16|Blk 56, #08-161 T...|Singapore|
|      2741|        Goh Chee Eng|             C| 45|     F|      25.91|Blk 267 Sembawang...|Singapore|
|      2820|    Ng Wee Hock John|             A| 56|     M|      66.22|Blk 417, Woodland...|Singapore|
+----------+--------------------+--------------+---+------+-----------+--------------------+---------+-
only showing top 20 rows
```

- Notice that the columns are truncated to limited sizes while the data for only 20 rows is shown.

- To see full set column width and more rows of data, you may use the following option of the **show** method. (i.e., up to 200 records are shown and column truncation set to false)

```
df.show(200, False)
```

Exploring Schema:

To explore the implicit schema that has been generated based on the header row and interpretation of the data type in the subsequent data row you may use the following statement:

```
df.printSchema()
```

The output so obtained is as below:

```
|-- CustomerID: integer (nullable = true)
|-- CustomerName: string (nullable = true)
|-- MemberCategory: string (nullable = true)
|-- Age: integer (nullable = true)
|-- Gender: string (nullable = true)
|-- AmountSpent: double (nullable = true)
|-- Address: string (nullable = true)
|-- City: string (nullable = true)
|-- CountryCode: string (nullable = true)
|-- ContactTitle: string (nullable = true)
|-- PhoneNumber: integer (nullable = true)
```

Setting Schema:

If you do not wish to use the implicit schema or if there are no header row in the file, then you can set your own definitions for column headers and the data types. There are a few ways of doing that, which you may explore from the api reference documentation.

One of the ways of doing is to define a schema and use that schema while reading the csv, as shown in codes below:

```
from pyspark import SparkContext, SparkConf
from pyspark.sql import *
from pyspark.sql.functions import *

def main():
    cnfg = SparkConf().setAppName("CustomerApplication").setMaster("local[2]")
    sc = SparkContext(conf=cnfg)
    spark = SparkSession(sc)

    inputFilePath = "D:\\workspace\\CustomerNoHdr.csv"

    custschema = StructType([
        StructField("Customerid", IntegerType(), True),
        StructField("CustName", StringType(), True),
        StructField("MemCat", StringType(), True),
```

```
        StructField("Age", IntegerType(), True),
        StructField("Gender", StringType(), True),
        StructField("AmtSpent", DoubleType(), True),
        StructField("Address", StringType(), True),
        StructField("City", StringType(), True),
        StructField("CountryID", StringType(), True),
        StructField("Title", StringType(), True),
        StructField("PhoneNo", StringType(), True)
        ])

    df.printSchema()

    df = ( spark.read
        .schema(schema=custschema)
        .csv(inputFilePath)  )

    df.show()

if __name__ == '__main__':
    main()
```

Note:

- In the above, I am reading the data from CustomerNoHdr.CSV file (which is a copy of the Customer CSV without the header row.
- The print schema will reflect the header names and data types that you have explicitly specified in the custschema definition
- The show() will reflect the data with the column header names that you have specified.

Example 2: Returning Selected Fields

Show only CustomerID, CustomerName and Age

- Note that when we return a dataset (customers in the above case), we get all columns.
- If we wish to return only selected fields, e.g, if you wish to only return CustomerID, CustomerName and Age and skip the other fields then you can specify the selected fields.

```
df.select(df["CustomerID"], df["CustomerName"], df["Age"]).show(200, False)
```

OR

```
df.select(CustomerID", "CustomerName", "Age").show(200, False)
```

OR

```
df.select(col("CustomerID"), col("CustomerName"), col("Age")).show(200, False)
```

SPARK QUERY LANGUAGE FUNCTIONS

Example 3: Retrieving data from the Customers Dataset & displaying sorted on Customer Name

```
df.orderBy("CustomerName").show()
```

- Use the **orderBy** method and the functional specification (argument) contains the column name expressed as a string.
- You will get the following output – notice the data is presented in ascending order of the names.

```
+----------+--------------------+--------------+---+------+-----------+---
|CustomerID|        CustomerName|MemberCategory|Age|Gender|AmountSpent|
+----------+--------------------+--------------+---+------+-----------+---
|      4567|         Abdul Zaidi|             A| 29|     M|      31.65|Blk
|      5489|        Ang Kim Beng|             A| 45|     M|      32.09|Blk
|      2669|   Boh Lee Ming Lynn|             C| 23|     F|      41.99|Blk
|      8080|      Chan Chin Fung|             B| 56|     M|      24.95|6 D
|      2270|   Chao Tah Jin Alex|             C| 22|     M|      87.07|Blk
|      7616|     Cheong Pei Sian|             B| 57|     F|      92.57|8 W
|      2828|         Cheryl Song|             C| 35|     F|      83.59|Blk
|      2983|          Cheryl Tan|             A| 37|     F|      13.99|Blk
|      1003|      Chew Teck Kuan|             C| 63|     M|      64.49|Blk
|      1818|    Chionh Choon Lee|             A| 57|     M|       7.13|Blk
|      5108|        Cho Wee Weng|             A| 50|     M|       24.8|Blk
|      6598|     Choey Choon Yen|             A| 28|     M|      88.01|30,
|      5168|      Constance Yong|             C| 46|     F|      30.64|Blk
|      8788|        Fu Shou Jeen|             A| 59|     M|      75.43|9 E
|      2741|        Goh Chee Eng|             C| 45|     F|      25.91|Blk
|      6969|         Jason Young|             C| 52|     M|      15.85|blk
|      2131|                 Jon|             A| 64|     F|      83.45|Blc
|      5568|        Kam Teng Mun|             A| 26|     M|      17.84|Blk
```

Variations, try the following:

Getting data in descending sequence: **`orderBy(desc("CustomerName")`**

```
+----------+--------------------+--------------+---+
|CustomerID|CustomerName        |MemberCategory|Age|
+----------+--------------------+--------------+---+
|1001      |Wong Sook Huey      |A             |37 |
|1810      |Vanessa Ong         |C             |32 |
|2233      |Too Siew Hong       |B             |35 |
|1681      |Terence Lim         |C             |30 |
|5968      |Tan Wei Wei         |A             |48 |
|3845      |Tan Choon Heong     |A             |21 |
|2626      |Steven Teo          |A             |56 |
```

Getting sorted by member category first then on Customer Name:

```
orderBy("MemberCategory", "CustomerName").show()
```

Example 4: Selective retrieval

a. <u>Obtaining only those Customers whose MemberCategory is 'A'</u>

- Use the **df.filter** function as below

```
filter("MemberCategory = 'A'").show()
```

```
+----------+--------------------+--------------+---+
|CustomerID|CustomerName        |MemberCategory|Age|
+----------+--------------------+--------------+---+
|1000      |Lou Anna Tan        |A             |29 |
|1001      |Wong Sook Huey      |A             |37 |
|1634      |Sridharan Jayanthi  |A             |55 |
|1818      |Chionh Choon Lee    |A             |57 |
|2323      |Richard Kwan        |A             |26 |
|2345      |Ng Teck Kie Anthony |A             |56 |
|2626      |Steven Teo          |A             |56 |
|2688      |Kathleen Loh Swat Hong|A           |38 |
```

b. <u>Obtaining only those Customers whose MemberCategory is 'A' & their name starts with 'T'</u>

- Use the **df.filter** function as below

```
filter("MemberCategory = 'A' AND CustomerName LIKE 'T%'").show()
```

```
+----------+--------------+--------------+---+
|CustomerID|CustomerName  |MemberCategory|Age|
+----------+--------------+--------------+---+
|3845      |Tan Choon Heong|A             |21 |
|5968      |Tan Wei Wei   |A             |48 |
+----------+--------------+--------------+---+
```

Example 5: Combining functions

In this example we will show the combination of filtering and sorting.

Retrieve all Customers who belongs to MemberCategory A and present in ascending order of names.

You will need to use multiple functions in succession as below.

```
df.filter("MemberCategory = 'A'").orderBy("CustomerName").show(300,False)
```

```
+----------+----------------------+--------------+---+
|CustomerID|CustomerName          |MemberCategory|Age|
+----------+----------------------+--------------+---+
|4567      |Abdul Zaidi           |A             |29 |
|5489      |Ang Kim Beng          |A             |45 |
|2983      |Cheryl Tan            |A             |37 |
|1818      |Chionh Choon Lee      |A             |57 |
|5108      |Cho Wee Weng          |A             |50 |
```

AGGREGATION AND SIMPLE STATISTICAL QUERY EXAMPLES

Example 6: Getting a count

Obtaining the number of A category members.

```
count = df.filter("MemberCategory = 'A'").count()
print(count)
```

Run the program to get the count *(in my database I got a result 26)*

Example 7: Getting the Sum of a field

Obtaining the Total Amount earned (i.e., sum of AmountSpent by all customers).

```
tot = df.agg(sum("AmountSpent")).first()[0]
print(tot)
```

Running this I got and output of 2486.3099.

Example 8: Getting the Sum of a field with condition on the rows to use

Obtaining the Total Amount earned from A category members (i.e., sum of AmountSpent by all customers whose MemberCategory is A).

```
tot = df.filter("MemberCategory ='A'").agg(sum("AmountSpent")).first()[0]
print(tot)
```

Running this I got and output of 1194.66

Example 9: Getting the Average of a field

Obtaining the Average Age of customers

```
tot = df.agg(avg("Age")).first()[0]
print(tot)
```

Running this I got and output of 41.82

COMPLEX & FURTHER STASTICAL QUERIES

The example on Group by where sub-totals are obtained may be a good example to mix both.

Example 10: Grouping & Subtotals based on single parameter

Obtaining Total Amount Spent for each all customers in each Member Category

```
df.groupBy("MemberCategory").sum("AmountSpent").show(200,False)
```

The output is shown below:

```
+--------------+------------------+
|MemberCategory|sum(AmountSpent)  |
+--------------+------------------+
|B             |500.40999999999997|
|C             |791.24            |
|A             |1194.66           |
+--------------+------------------+
```

Example 11: Grouping & Subtotals based on multiple parameters

Obtaining Total Amount Spent by customers based on their Member Category & Gender

```
df.groupBy("MemberCategory","Gender")
              .sum("AmountSpent")
              .orderBy("MemberCategory","Gender")
              .show(200,False)
```

The output is shown below:

```
+--------------+------+-----------------+
|MemberCategory|Gender|sum(AmountSpent) |
+--------------+------+-----------------+
|A             |F     |472.2199999999999|
|A             |M     |722.4399999999998|
|B             |F     |216.48           |
|B             |M     |283.93           |
|C             |F     |401.51           |
|C             |M     |389.73           |
+--------------+------+-----------------+
```

Note: Data case sensitivity may affect the grouping, so you may need to ensure that this data is clean.

Example 12: ROLL UP function *(vs Group By)*

```
df.rollup("MemberCategory","Gender")
                .sum("AmountSpent")
                .orderBy("MemberCategory","Gender")
                .show(200,False)
```

The output is shown below:

```
+--------------+------+------------------+
|MemberCategory|Gender|sum(AmountSpent)  |
+--------------+------+------------------+
|null          |null  |2486.309999999999 |
|A             |null  |1194.66           |
|A             |F     |472.2199999999999 |
|A             |M     |722.4399999999998 |
|B             |null  |500.40999999999997|
|B             |F     |216.48            |
|B             |M     |283.93            |
|C             |null  |791.24            |
|C             |F     |401.51            |
|C             |M     |389.73            |
+--------------+------+------------------+
```

Observe that there are more rows now. The rows with "null" represent the ***totals*** of all the grouping divisions. For instance the value reflected in **B-Null** is the sum of **B-M** and **B-F**

Example 13: The CUBE function instead of ROLL UP

```
df.cube("MemberCategory","Gender")
                .sum("AmountSpent")
                .orderBy("MemberCategory","Gender")
                .show(200,False)
```

The output is shown below:

```
+--------------+------+------------------+
|MemberCategory|Gender|sum(AmountSpent)  |
+--------------+------+------------------+
|null          |null  |2486.309999999999 |
|null          |F     |1090.21           |
|null          |M     |1396.1000000000001|
|A             |null  |1194.66           |
|A             |F     |472.2199999999999 |
|A             |M     |722.4399999999998 |
|B             |null  |500.40999999999997|
|B             |F     |216.48            |
|B             |M     |283.93            |
|C             |null  |791.24            |
|C             |F     |401.51            |
|C             |M     |389.73            |
+--------------+------+------------------+
```

Observe that while in the case of rollup the MemberCategory was taken as primary and the total for only the sub-category was performed, the cube is multidimensional. This means computation of totals is performed from both MemberCategory as primary and Gender as primary. So while rollup function provides only totals for MemberCategory (i.e., sum of all Gender for each Member Category), the cube function provides totals for all MemberCategory (i.e. sum by Gender for each Member Category) AS WELL AS the totals of all Gender (i.e., sum by all MemberCategory for each Gender).

Example 14: Getting the Standard Deviation of a field

Obtaining the Standard Deviation of the Amounts that the customers spent.

```
std = df.agg(stddev_pop("AmountSpent")).first()[0]
print(std)
```

Running this I got and output of 28.895954726570288

Note: If you use stddev_samp instead, you will get the sample standard deviation which in my case returned a number *29.189322194310456*

Example 15: Getting the Skewness of a field

Obtaining the skewness of the Amounts that the customers spent.

```
skw = df.agg(skewness("AmountSpent")).first()[0]
print(skw)
```

Running this I got and output of -0.09241609965633672

NOTE: You may try more statistical functions. For quick reference, the key aggregate statistical functions are listed in the appendix

Example 16: Getting the Common Statistics of a few fields in one function call

If you desire to obtain the most common statistical data of certain fields you may use the **describe** function as depicted below.

```
df.describe("MemberCategory", "Gender", "AmountSpent", "Age").show()
```

The output is shown below:

```
+-------+--------------+------+------------------+------------------+
|summary|MemberCategory|Gender|       AmountSpent|               Age|
+-------+--------------+------+------------------+------------------+
|  count|            50|    50|                50|                50|
|   mean|          null|  null| 49.72619999999998|             41.82|
| stddev|          null|  null|29.189322194310456|13.625740945964393|
|    min|             A|     F|              2.98|                21|
|    max|             C|     M|             99.52|                64|
+-------+--------------+------+------------------+------------------+
```

MULITIPLE ENTITIES

Example 17: Joining two data sources (CSV) and retrieving data from both

We will create two dataframes to load Customers and Country data. The country table which has been derived on normalising the country specific fields has a countrycode that maps to the countrycode field of the customers data.

The following code (presented in Example 1) is augmented to include two dataframes, df1 and df2.

```
# imports here
def main():
    cnfg = SparkConf().setAppName("CustomerApplication").setMaster("local[2]")
    sc = SparkContext(conf=cnfg)
    spark = SparkSession(sc)

    customerFilePath = "D:\\workspace\\Customer.csv"
    countryFilePath = "D:\\workspace\\Country.csv"

    dfCustomer =  ( spark.read
                    .option("header", "true")
                    .option("inferSchema", "true")
                    .csv(customerFilePath)   )

    dfCountry = ( spark.read
                 .option("header", "true")
                 .option("inferSchema", "true")
                 .csv(countryFilePath)          )

    joinDF = dfCustomer.join(dfCountry, "CountryCode")
    (  joinDF.select("CustomerID", "CustomerName", "CountryCode",
              "CountryName", "Currency", "TimeZone")
                .show(300, False)   )     )

if __name__ == '__main__':
    main()
```

The output is shown below:

```
+----------+---------------------+-----------+-----------+--------+--------+
|CustomerID|CustomerName         |CountryCode|CountryName|Currency|TimeZone|
+----------+---------------------+-----------+-----------+--------+--------+
|1000      |Lou Anna Tan         |GER        |Germany    |EUR     |2       |
|1001      |Wong Sook Huey       |SIN        |Singapore  |SGD     |8       |
|1002      |Ng Choon Seng        |CAN        |Canada     |CND     |-6      |
|1003      |Chew Teck Kuan       |SIN        |Singapore  |SGD     |8       |
|1111      |Steven Ou            |BRA        |Brazil     |BRL     |-3      |
|1634      |Sridharan Jayanthi   |SIN        |Singapore  |SGD     |8       |
|1681      |Terence Lim          |SLR        |Srilanka   |Srp     |0       |
|1810      |Vanessa Ong          |SIN        |Singapore  |SGD     |8       |
|1811      |Koh Ting Ting        |SIN        |Singapore  |SGD     |8       |
|1818      |Chionh Choon Lee     |SIN        |Singapore  |SGD     |8       |
|2131      |Jon                  |SWZ        |Switzerland|SFR     |2       |
```

In the above table the first two fields are from the Customers Data, while the last three are from Country data – the right side fields are joined on the common field CountryCode.

Note: You may wish to try other types of joins such as outer join etc. for your own interest.

RETRIEVING DATA FROM OTHER DATA SOURCE TYPES

Example 18: Retrieving Data from Json Source.

Using the following code segment should work:

```
inputFilePath = "D:\\workspace\\Producers.json"

df = ( spark.read
          .option("header", "true")
          .option("inferSchema", "true")
          .json(inputFilePath)  )

df.show(200, False)
```

Please note that all the lines of code are similar to the CSV example. For Json file we use the **read.json** – that is the only difference.

Example 19: Retrieving Data from MySQL Source.

The following code should work:

```
from pyspark import SparkContext, SparkConf
from pyspark.sql import *
from pyspark.sql.functions import *
from pyspark.sql.types import *

def main():
    cnfg = SparkConf().setAppName("CustomerApplication").setMaster("local[2]")
    cnfg.set("spark.jars","D:\\mysql-connector-java\\mysql-connector-java-5.1.49.jar")
    sc = SparkContext(conf=cnfg)
    spark = SparkSession(sc)

    df = ( spark.read.format("jdbc")
            .options(url="jdbc:mysql://localhost/videoshop",
                    driver="com.mysql.jdbc.Driver",
                    dbtable="Movies",
                    user="venkat",
                    password="P@ssw0rd1" ).load())

    df.show(200,False)

if __name__ == '__main__':
    main()
```

Note: Please ensure that you download the **mysql-connector-java-5.1.49.jar** into an appropriate folder and reference the path it in the configuration. You should also use the database, user and password as per your local installation.

WRITE OPERATIONS

Example 20: Saving a CSV file

To demonstrate the write operations this simple example reads the Country.csv CSV file and write it to a json file.

We use the df.write method to create a new json file.

```
from pyspark import SparkContext, SparkConf
from pyspark.sql import *
from pyspark.sql.functions import *
from pyspark.sql.types import *

def main():
    cnfg = SparkConf().setAppName("CustomerApplication").setMaster("local[2]")
    sc = SparkContext(conf=cnfg)
    spark = SparkSession(sc)

    inputFilePath = "D:\\scala\\workspace\\Country.csv"

    df = ( spark.read
            .option("header", "true")
            .option("inferSchema", "true")
            .csv(inputFilePath) )

    df.write.json("D:\\CountryOUT")

    if __name__ == '__main__':
        main()
```

Please note the output path is a folder (not existing) – *not a json file name.*

A new JSON file gets created in the specified folder with a system generated file name.

In my case the file was named: **part-00000-2ee2d8b6-169a-4f48-a503-8b6a2fdedab0-c000.json**

The file content was as shown below:

```
{"CountryCode":"ARG","CountryName":"Argentina","Currency":"ARS","TimeZone":"-3"}
{"CountryCode":"AUS","CountryName":"Australia","Currency":"AUD","TimeZone":"10"}
{"CountryCode":"BEL","CountryName":"Belgium","Currency":"EUR","TimeZone":"1"}
{"CountryCode":"BRA","CountryName":"Brazil","Currency":"BRL","TimeZone":"-3"}
{"CountryCode":"CAN","CountryName":"Canada","Currency":"CND","TimeZone":"-6"}
...
...
{"CountryCode":"VEN","CountryName":"Venezuela","Currency":"NULL","TimeZone":"NULL"}
```

Trial options for your further understanding:

1. You may read a json file (eg: Producers.CSV) and output a CSV file.

2. You may use a large file (can download from Kaggle website) and re run the above program. If the file is larger than 128MB, you will notice that the file is saved in multiple parts (technically in a distributed cluster this will be sharded). I ran US accidents csv data and saved to a json and got it in 7 shards.

3. You may try reading back the data from multiple shards (parts). For instance, I had written the US Accidents csv data into Json as 7 file shards. To read all the parts in the folder created, we need to use the file path as: **"D:\myFolder*.json"**
   The above will read all the parts – you may test it with a count check to ensure all the files' data has been consolidated into the data frame.

WORKING ON HDFS FILE SYSTEM *(linux workshop only)*

For learning purposes of Spark SQL, we had used local file system. The programs we wrote will work even if the files were deployed in HDFS. For your testing purpose we have loaded the files in the following HDFS directory (already packaged in your VM image):

/user/cloudera/

To access data from HDFS, you may have use the following URI (file path) in your program. All other statements will remain same.

```
customerPath = "hdfs://quickstart.cloudera/user/cloudera/Customer.csv"
```

Note: To rund HDFS you will need a linux system (or Virtual Machine). At present this is not available in windows based workshop. A more comprehensive workshop on working with HDFS file is included in the lesson on Hadoop Introduction. This instruction confines to the minimal input concerning the retrieval of data stored in HDFS only.

PRACTICE EXERCISES

Using Dataframes and SparkSQL and working on **Movies.CSV** write Spark SQL for the following:

A. *Data retrieval using Spark SQL*

1. Retrieve all data from the Movies table.
2. Retrieve all Movies and display the data in ascending order of Movie Title.
3. Retrieve all R rated movies. You should display only the Video Code, Movie Title, and Movie Type.
4. Retrieve all Science Fiction movies produced by Warner.

B. *Aggregation and Statistical Queries*

5. Determine the average rental price of all movies. Explore the variance, standard deviation and skewness of this population.
6. Find the total stock of movies grouped by Movie Type and Rating (two fields).

   Use the RollUp function and build the cube function to determine the dimensional totals for the above grouping.

C. *JSON files*

7. Read data from the JSON file **Producers.JSON** and display all records.
8. Modify the above query to display those producers who are located in UK.
9. Read the **Movies.CSV** into a Dataframe and write it as a JSON format file. Open Json file to observe the conversion schema.

D. *Multiple Entities Joining and multiple formats joined in a DataFrame*

10. Retrieve all movies produced by Walt Disney Studios. You should display the Producer Name, Location, Movie Title and Movie Type. *(Note: The first two fields are from Producer.JSON and the other two fields are from Movies.CSV. The join field is ProcuderCode field. If you have issues, you may change ProducerID column in the CSV to same as in JSON, viz. ProducerCode)*

APPENDIX A

SELECTED STATISTICAL AGGREGATE FUNCTION

Function	Meaning
approx_count_distinct(columnName: String)	Aggregate function: returns the approximate number of distinct items in a group.
avg(columnName: String)	Aggregate function: returns the average of the values in a group.
avg(e: Column): Column	Aggregate function: returns the average of the values in a group.
collect_list(columnName: String): Column	Aggregate function: returns a list of objects with duplicates.
collect_set(columnName: String): Column	Aggregate function: returns a set of objects with duplicate elements eliminated.
corr(columnName1: String, columnName2: String): Column	Aggregate function: returns the Pearson Correlation Coefficient for two columns.
count(columnName: String): TypedColumn[Any, Long]	Aggregate function: returns the number of items in a group.
countDistinct(columnName: String, columnNames: String*): Column	Aggregate function: returns the number of distinct items in a group.
covar_pop(columnName1: String, columnName2: String): Column	Aggregate function: returns the population covariance for two columns.
covar_samp(columnName1: String, columnName2: String): Column	Aggregate function: returns the sample covariance for two columns.
first(columnName: String): Column	Aggregate function: returns the first value of a column in a group.
kurtosis(columnName: String): Column	Aggregate function: returns the kurtosis of the values in a group.
last(columnName: String): Column	Aggregate function: returns the last value of the column in a group.
max(columnName: String): Column	Aggregate function: returns the maximum value of the column in a group.
mean(columnName: String): Column	Aggregate function: returns the average of the values in a group.
min(columnName: String): Column	Aggregate function: returns the minimum value of the column in a group.
skewness(columnName: String): Column	Aggregate function: returns the skewness of the values in a group.
stddev(columnName: String): Column	Aggregate function: alias for stddev_samp.
stddev_pop(columnName: String): Column	Aggregate function: returns the population standard deviation of the expression in a group.
stddev_samp(columnName: String): Column	Aggregate function: returns the sample standard deviation of the expression in a group.
sum(columnName: String): Column	Aggregate function: returns the sum of all values in the given column.
sumDistinct(columnName: String): Column	Aggregate function: returns the sum of distinct values in the expression.
var_pop(columnName: String): Column	Aggregate function: returns the population variance of the values in a group.
var_samp(columnName: String): Column	Aggregate function: returns the unbiased variance of the values in a group.
variance(columnName: String): Column	Aggregate function: alias for var_samp.