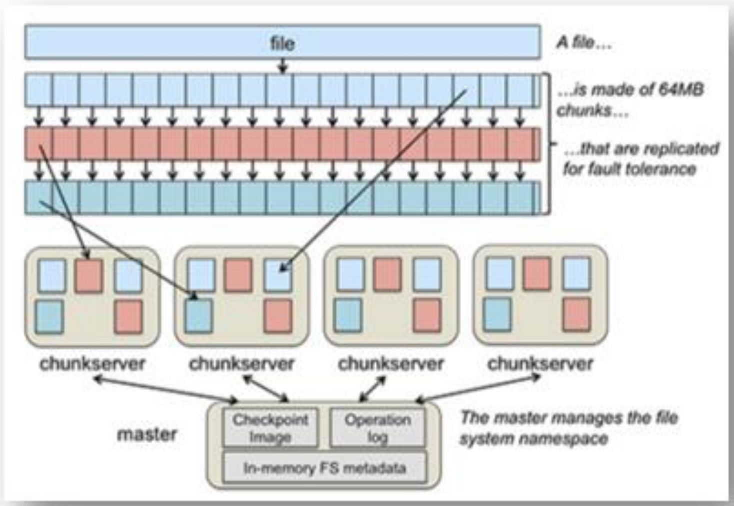
Part (1) Definitions

1. What do you understand by a distributed file system?

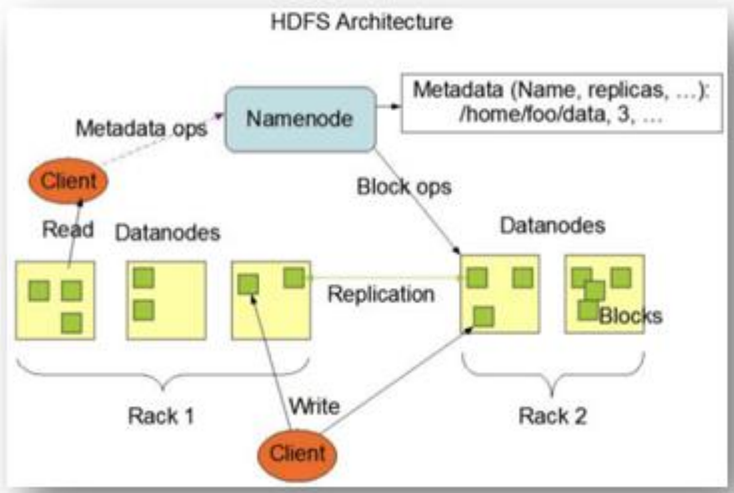
Distributed file system (DFS) – is a file system that is distributed on multiple file servers or locations. the users can share files and storage resources as they do in the local computer. also, allowing programmers to access files from any network. A DFS manages a set of dispersed storage devices. [3]

Briefly describe with examples, two implementations of a distributed file system?

a. **Google File System,** a distributed file system for large distributed data-intensive applications. It is a DFS developed by Google. It is designed to provide reliable access to data using large clusters of commodity hardware, efficient. it is optimized to run on computing clusters, the nodes of which consist of cheap [2]


file
A file...
...is made of 64MB chunks...
...that are replicated for fault tolerance
chunkserver
chunkserver
chunkserver
chunkserver
master
Checkpoint Image
Operation log
In-memory FS metadata
The master manages the file system namespace

b. **HDFS** is a File System. It is designed to store very large files across machines in a large cluster.HDFS is also designed to provide streaming of data clusters to the client application with high bandwidth. Due to its reliability & efficiency, it is widely used in Big data analytics like Hadoop MapReduce and spark.[1]



References:

[1]"**HDFS Architecture Guide**" https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html **by Dhruba Borthakur**

[2] "**Review on Distributed File System**" https://inthiraj1994.medium.com/review-on-existing-distributed-file-system-1d78ad0a9a5d **by Tharmakulasingham Inthirakumaaran** (accessed by **Jun 30)**

[3] "**Distributed File System (DFS) | microsoft Docs"**https://docs.microsoft.com/en-us/windows/win32/dfs/distributed-file-system-dfs-functions**(accessed by jan.6.2021)**

2. Briefly describe 3 features of Apache Hadoop Map-Reduce and 3 limitations associated with it when compared to Apache Spark?

- Map-Reduce Advantages:

    1- Scalability: Hadoop can store and distribute big datasets over many servers The servers utilized here are quite low-cost and can run in parallel. With the ability of adding new servers, the system's processing power may be increased as well.

    2- Faster Execution: Hadoop Map Reduce uses data locality, which means moving the code to the node where data resides instead of moving the data to the note, Thus, it makes the execution very fast because the code is much less in storage compared to the data

    3- Fault tolerant: as by default, 3 replicas of each block are stored across the cluster. So if any node goes down, data on that node can recover from the other node easily.

- Map-Reduce Limitations compared with Apache Spark:

    1- MapReduce cannot easily process large scale Machine Learning algorithms and complex analytics because iterative algorithms need to make multiple passes (10 – 20) over the data in the other hand using Apache spark is more efficient cause its 100x times faster than Hadoop Map-Reduce.

    2- **Not Easy to Use:** With Hadoop Map-Reduce you have to make and write everything by yourself but Apache spark makes writing big data applications much easier by having Spark API layers and High-level operators

    3- **Slow Processing speed:** Hadoop MapReduce persists data back to the disk after a map or reduce action which is very slow when compared to Apache Spark which processes data in random access memory (RAM) so its 100 times faster.

4- **No Real-Time Data Processing:** Apache Hadoop is designed for batch processing, which means it takes a large quantity of data, processes it, and outputs the result. Although batch processing is incredibly effective for processing large amounts of data, the result may be somewhat slow. Apache Spark, on the other hand, enables stream processing. Using the Apache API, stream processing requires continuous data input and output.

Resources:

Lecture Notes

Team, D. (2019, March 7). *Limitations of Hadoop*. DataFlair. Retrieved November 6, 2021, from https://data-flair.training/blogs/13-limitations-of-hadoop/

Pedamkar, P. (2021, November 6). *What-is-mapreduce*. Educba. Retrieved November 6, 2021, from https://www.educba.com/what-is-mapreduce/

Team, T. (2021, July 6). *Salient Features Of MapReduce – Importance of MapReduce*. TechVidvan. Retrieved November 6, 2021, from https://techvidvan.com/tutorials/hadoop-mapreduce-features/

3. Describe the low-level and high-level APIs in Apache Spark. What differentiates them and when do you use one over the other?

Low-level APIs provide visibility to the cluster partitions

An example of the low-level API is the RDDs. RDDs are the basic component of spark programming. RDDs can work on a parallel operation then it's separated among cluster nodes. RRDs can be created by spark context text file or by spark context parallelization. the RDDs transformations are

map(), filter(), distinct(), union(), intersection(), subtract(), sample()

Those functions are applied to all elements of the collection

and the pair RDD transformations is another kind of transformation that contains key/value pairs like a tuple of data.

the pair RDD transformation are:

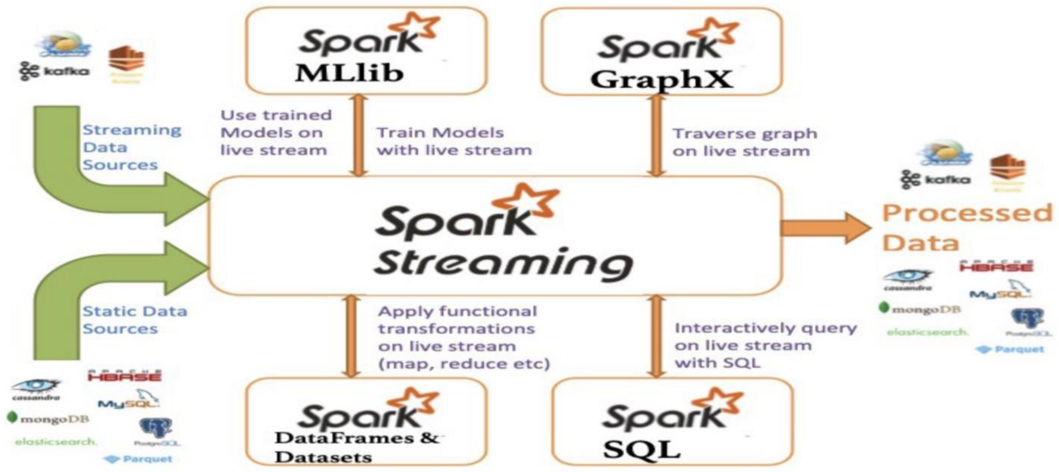
groupByKey(), reduceByKey(), sortByKey(), subtractByKey(), countByKey(), join()

and finally, the RDDs do not return the final results without using an "Action" function. these functions are

collect(), take(), count(), top(), reduce(),first(), sum(), aggregate()

Leverage **high-level APIs** to provide a broad range of functionalities.

An example of high-level API in apache spark are (Structure steaming, Advanced analytics, libraries and the ecosystems). In the high-level API spark supports high level programming languages such (Scala, Python, R, Java). Also in the high-level API, we can find the spark components such as (Spark core component, Spark SQL component, MLIB, Spark streaming, GraphX)


Spark MLlib
Spark GraphX
Streaming Data Sources
Use trained Models on live stream
Train Models with live stream
Traverse graph on live stream
Spark Streaming
Processed Data
Static Data Sources
Apply functional transformations on live stream (map, reduce etc)
Interactively query on live stream with SQL
Spark DataFrames & Datasets
Spark SQL
kafka
cassandra
HBASE
MySQL
mongoDB
elasticsearch
PostgreSQL
Parquet

DataFrame API is used with high-level APIs - Spark SQL, Spark Streaming, MLlib and GraphX. Structured Streaming is meant to be an easier-to-use and higher-performance evolution of Spark Streaming's DStream API. Higher-level streaming API built from the ground up on Spark's Structured APIs.

A high-level Spark API

- It is micro-batch oriented
- Treats the stream as a series of batches of data
- New batches are created at regular time intervals
- The size of the time interval is called the batch interval (typically between 500 ms to several seconds)
- No support for event time & continuous execution modes

Resources:

Lecture Notes

4. Describe the following Apache Spark terms with examples

a. Immutability in Spark.
b. Lazy Evaluation and its impact on Spark performance.
c. How is SparkSession different from SparkContext.
d. Spark MLlib Transformers, Estimators, and Evaluators.

a- Immutability in Spark: In Spark, data APIs can not be changed after they are created, making them immutable to change will reduce the complexity of the synchronization and be easier in parallelization, also it will make caching and sharing easier.Changes are allowed during the transformations phase.The tradeoff is that we will need to copy the data instead of changing it directly.

b- **Lazy Evaluation:** Delaying the data execution until an action trigger transformations.
Types of transformation actions:
- Collecting the data into objects.
- Using the console to view the data.
- Write the output to data sources.

Lazy Evaluation advantages:
- Reducing the RDD operations.
- Increasing the total speed.
- Easy on management.

c- **SparkContext:** Spark context was used widely as the starting point before spark 2.0 and it needs to be created for any spark interaction. Lots of operations that we use in spark come from SparkContext like broadcast variables, parallelize, and accumulators, also Lower level APIs such as **RDDs** still require a SparkContext.
Examples:
- SQL Context.
- Streaming Context.
- Hive Context.

d- MLlib Transformers: Functions that transform the collected data into other data forms that could be used in training ML models.
Transformers examples:
- StopWordsRemover.
- SQLTransformer.
- Tokenizer Transformer.

MLlib Estimators: Algorithms that allow people to train different models from the collected data.
Estimators examples:
- LogisticRegression.
- K-Mean.
- decision tree.

MLlib Evaluators: Allow us to see the performance of the models according to the metrics that we want.
Evaluators examples:
- RegressionEvaluator.
- MulticlassClassificationEvaluator.
- BinaryClassificationEvaluator.

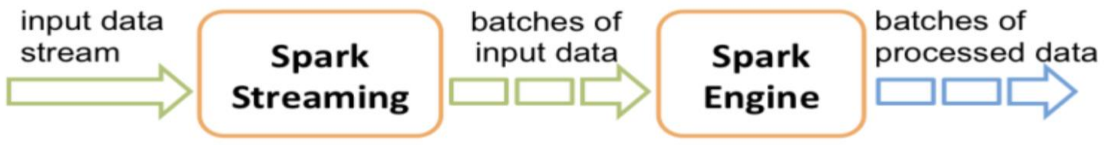
Resources:

M. (2021a). *Evaluators · Spark*. Gitbooks. https://mallikarjuna_g.gitbooks.io/spark/content/spark-mllib/spark-mllib-evaluators.html
M. (2021b). *ML Pipeline Components — Transformers · Spark*. Gitbooks. https://mallikarjuna_g.gitbooks.io/spark/content/spark-mllib/spark-mllib-transformers.html
N. (2020, July 26). *Post author: NNK*. Spark by {Examples}. https://sparkbyexamples.com/spark/sparksession-vs-sparkcontext/
Team, D. (2018, November 21). *Lazy Evaluation in Apache Spark – A Quick guide*. DataFlair. https://data-flair.training/blogs/apache-spark-lazy-evaluation/

5. Describe briefly – with examples - how Spark Streaming differs from Spark Structured Streaming?

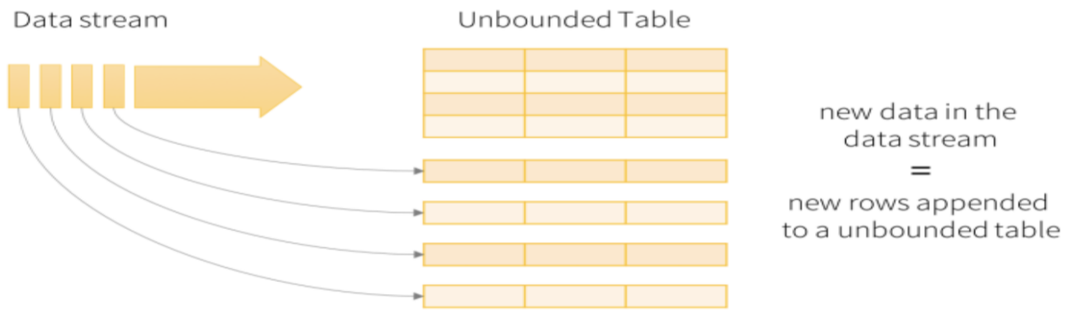
Spark streaming works on something which we call a micro-batch.

That means it will have tons of records for example one million then the micro-batch will trigger after every batch duration. Operations on the DStream translates to operations on the underlying RDDs.Discretized Stream or DStream is the basic abstraction provided by Spark Streaming (managed through the StreamingContext)



Structured Streaming works on the same architecture of polling the

data after some duration.in structured streaming, there is no concept of the batch. The data is moved based on the trigger interval the data is flowing continuously. Supports higher-level optimizations, event time, and continuous processing


Data stream
Unbounded Table
new data in the data stream
=
new rows appended to a unbounded table

Data stream as an unbounded table

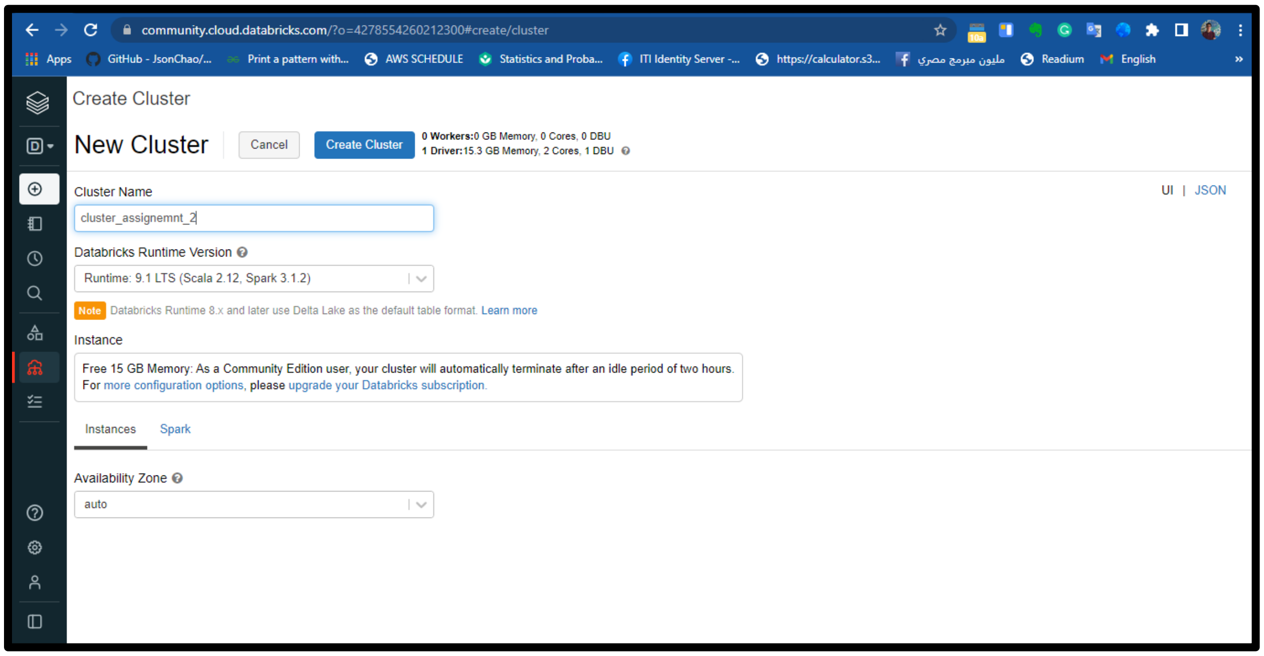
Resources:

- Lecture Notes

Part 2 – Spark Examples

1. Data Transformation Pipelines:

Cluster Setup



Create Cluster

New Cluster

0 Workers:0 GB Memory, 0 Cores, 0 DBU
1 Driver:15.3 GB Memory, 2 Cores, 1 DBU

Cluster Name: cluster_assignemnt_2

Databricks Runtime Version: Runtime: 9.1 LTS (Scala 2.12, Spark 3.1.2)

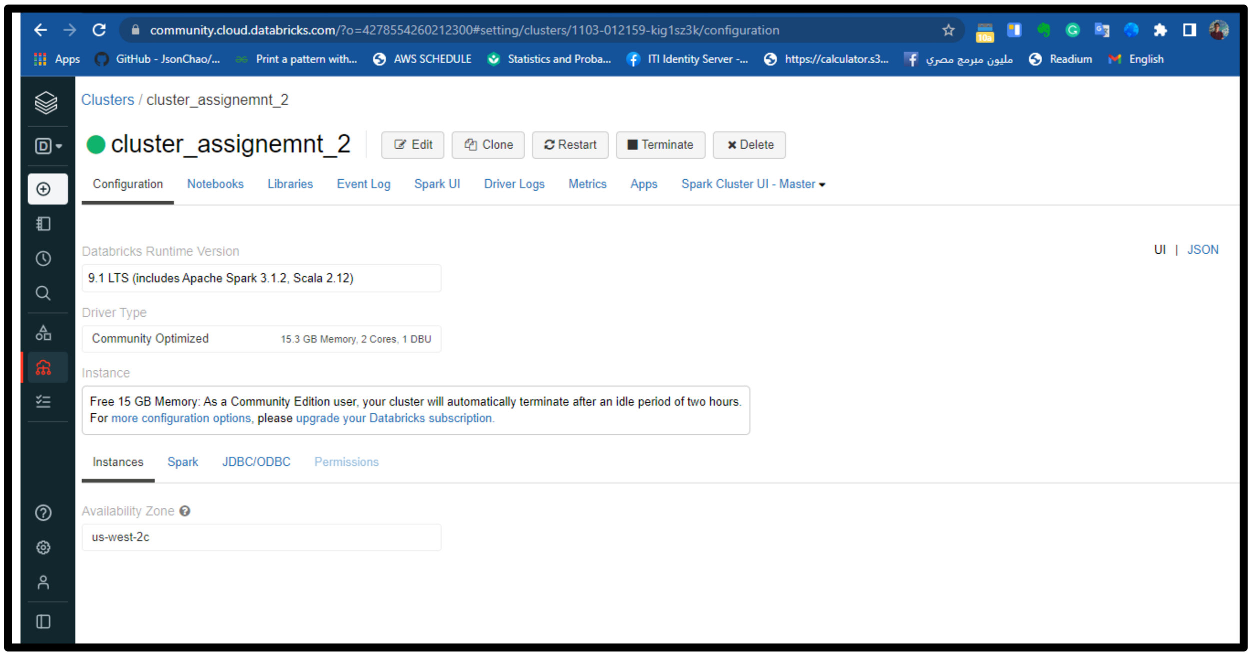
Note Databricks Runtime 8.x and later use Delta Lake as the default table format. Learn more

Instance

Free 15 GB Memory: As a Community Edition user, your cluster will automatically terminate after an idle period of two hours.
For more configuration options, please upgrade your Databricks subscription.

Availability Zone: auto

Cluster run Successfully



Clusters / cluster_assignemnt_2

cluster_assignemnt_2

Databricks Runtime Version: 9.1 LTS (includes Apache Spark 3.1.2, Scala 2.12)

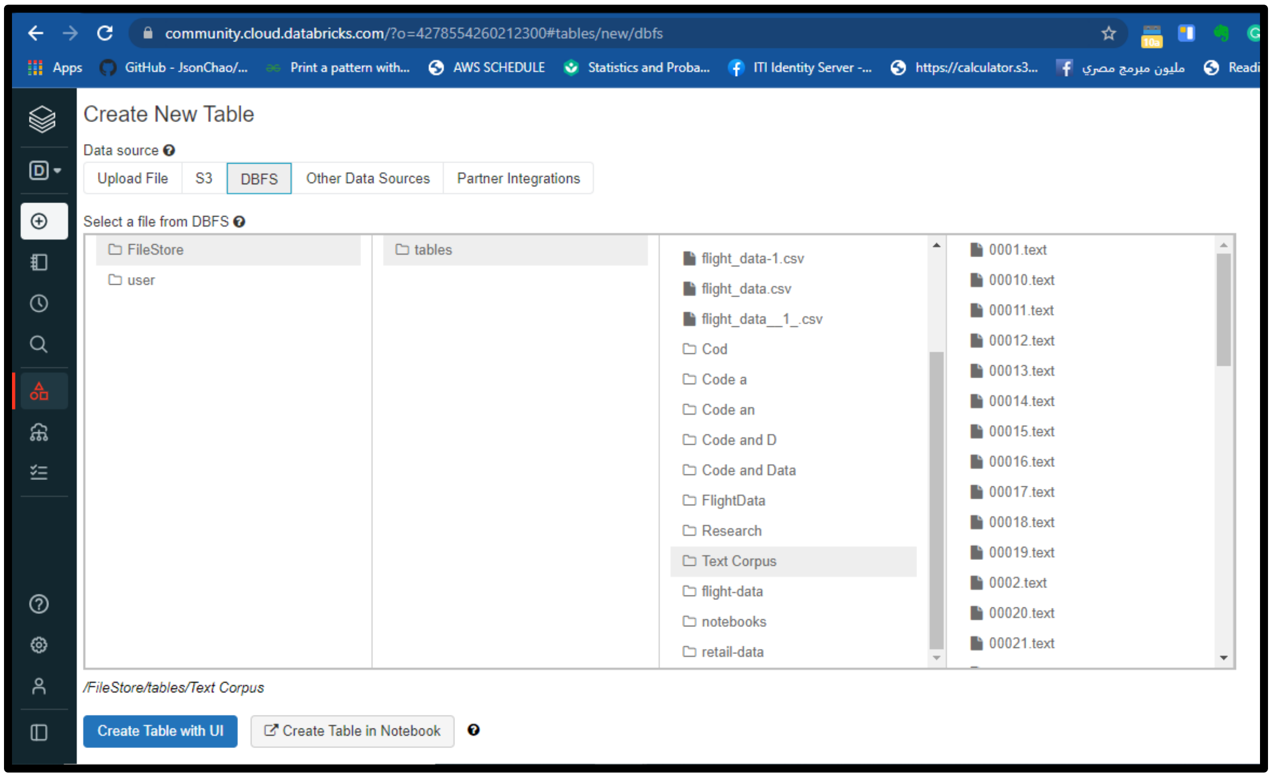
Driver Type: Community Optimized 15.3 GB Memory, 2 Cores, 1 DBU

Instance

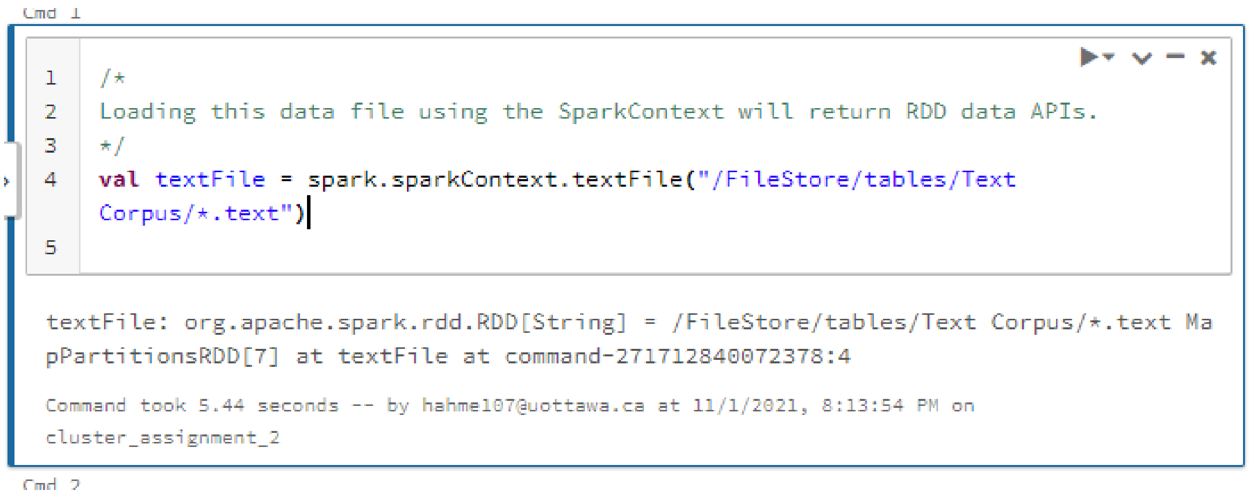
Free 15 GB Memory: As a Community Edition user, your cluster will automatically terminate after an idle period of two hours.
For more configuration options, please upgrade your Databricks subscription.

Availability Zone: us-west-2c

a) Uploaded the files to your DBFS table space.



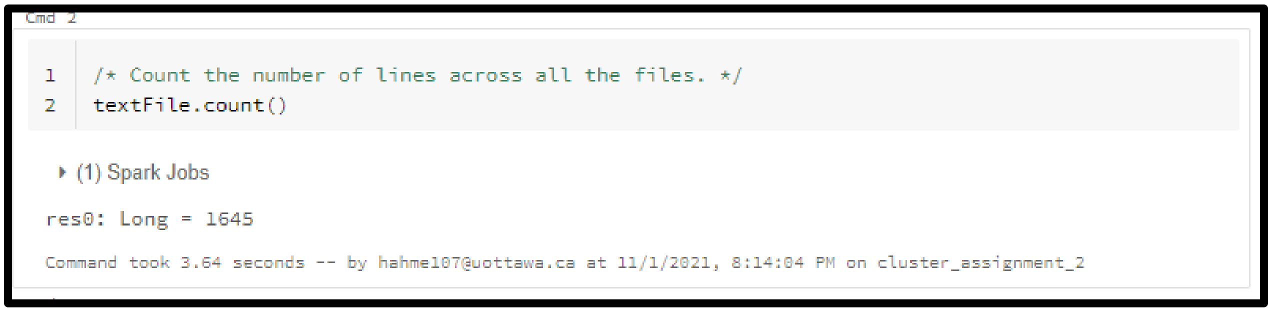
b) Use Spark Scala to load your data into an RDD.



```
/*
Loading this data file using the SparkContext will return RDD data APIs.
*/
val textFile = spark.sparkContext.textFile("/FileStore/tables/Text Corpus/*.text")
```

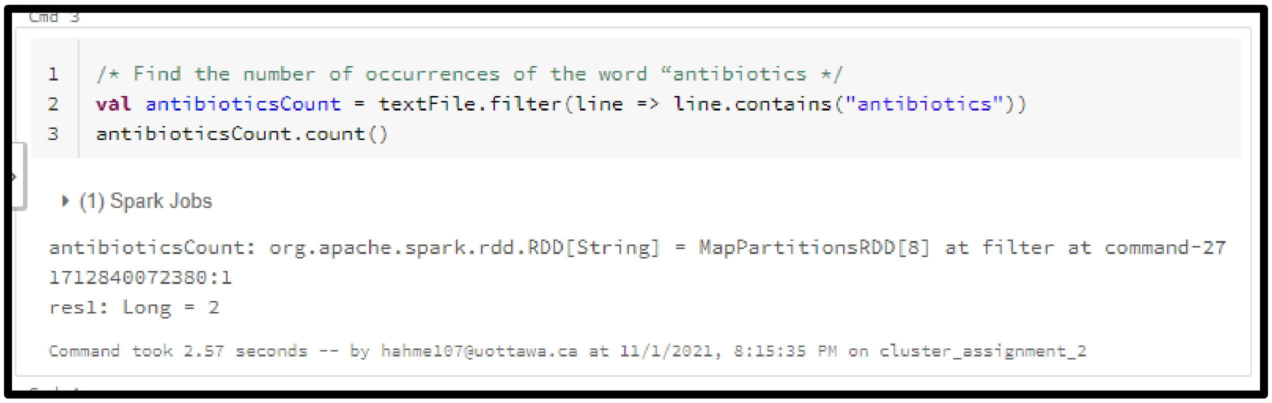
```
textFile: org.apache.spark.rdd.RDD[String] = /FileStore/tables/Text Corpus/*.text MapPartitionsRDD[7] at textFile at command-271712840072378:4
Command took 5.44 seconds -- by hahme107@uottawa.ca at 11/1/2021, 8:13:54 PM on cluster_assignment_2
```

c) Count the number of lines across all the files.



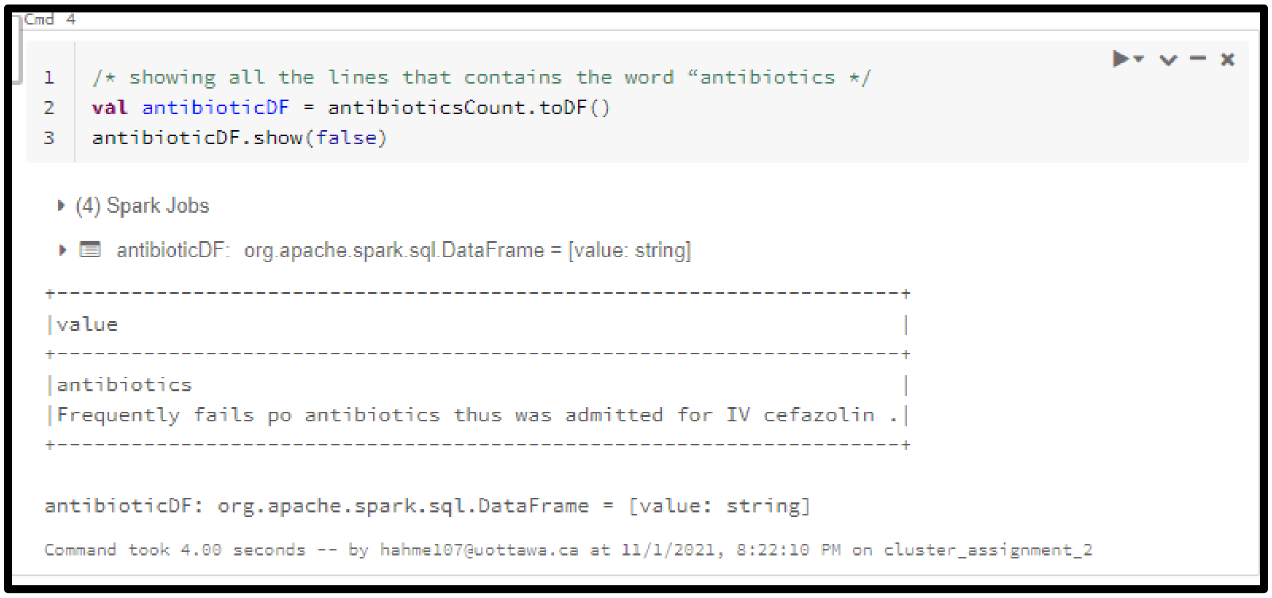
```
Cmd 2
/* Count the number of lines across all the files. */
textFile.count()

(1) Spark Jobs
res0: Long = 1645
Command took 3.64 seconds -- by hahme107@uottawa.ca at 11/1/2021, 8:14:04 PM on cluster_assignment_2
```

d) Find the number of occurrences of the word “antibiotics”



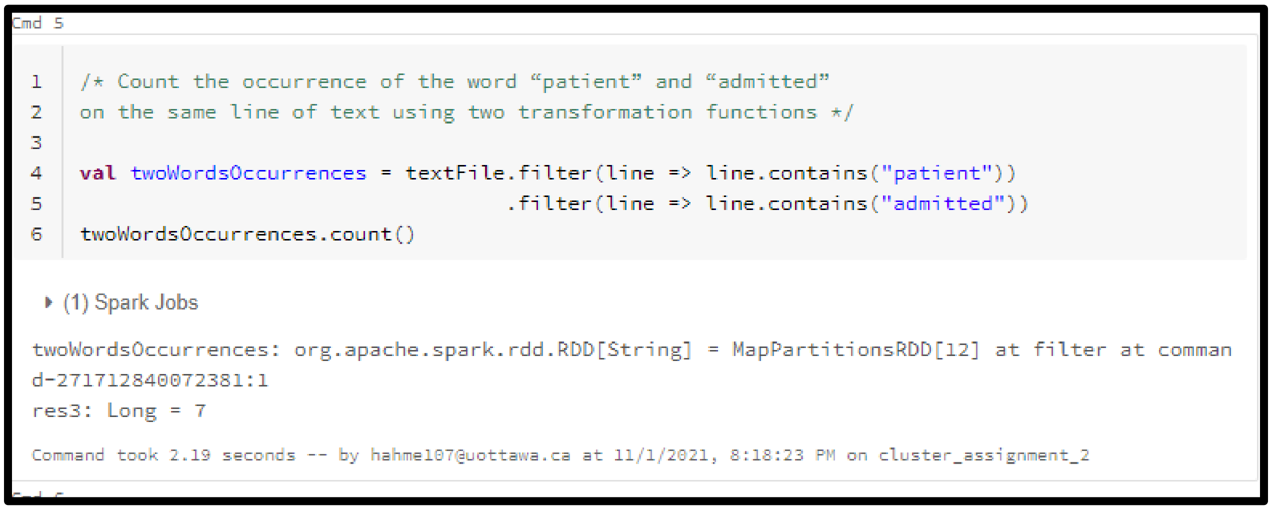
```
Cmd 3
/* Find the number of occurrences of the word “antibiotics */
val antibioticsCount = textFile.filter(line => line.contains("antibiotics"))
antibioticsCount.count()

(1) Spark Jobs
antibioticsCount: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[8] at filter at command-27
1712840072380:1
res1: Long = 2
Command took 2.57 seconds -- by hahme107@uottawa.ca at 11/1/2021, 8:15:35 PM on cluster_assignment_2
```

Show the occurrences of the word “antibiotics”



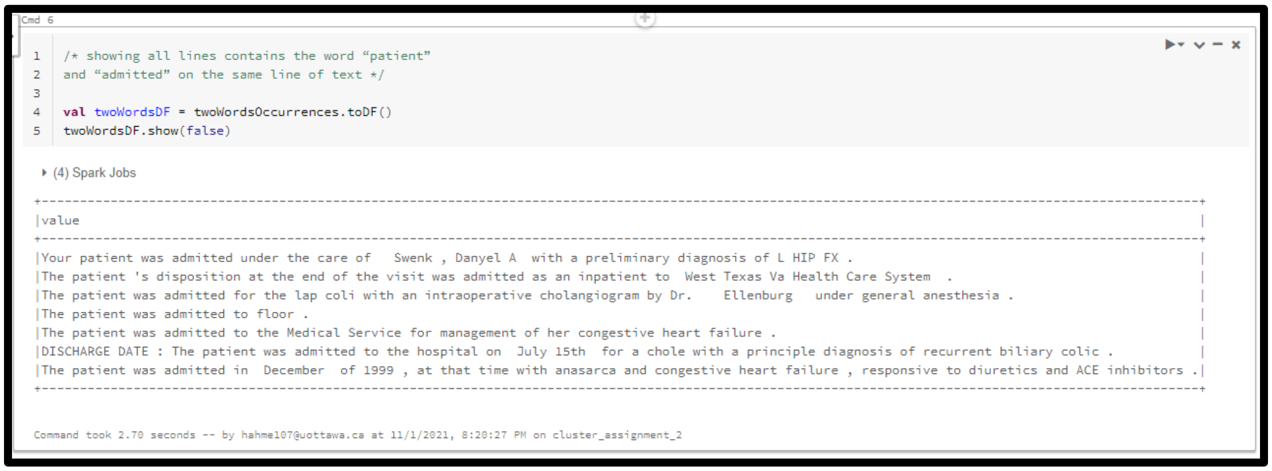
```
Cmd 4
/* showing all the lines that contains the word “antibiotics */
val antibioticDF = antibioticsCount.toDF()
antibioticDF.show(false)

(4) Spark Jobs
antibioticDF: org.apache.spark.sql.DataFrame = [value: string]
+---------------------------------------------------------------------+
|value                                                                |
+---------------------------------------------------------------------+
|antibiotics                                                          |
|Frequently fails po antibiotics thus was admitted for IV cefazolin .|
+---------------------------------------------------------------------+

antibioticDF: org.apache.spark.sql.DataFrame = [value: string]
Command took 4.00 seconds -- by hahme107@uottawa.ca at 11/1/2021, 8:22:10 PM on cluster_assignment_2
```

e) Count the occurrence of the word "patient" and "admitted" on the same line of text. Please ensure that your code contains at least 2 transformation functions in a pipeline.



```
Cmd 5

/* Count the occurrence of the word "patient" and "admitted"
on the same line of text using two transformation functions */

val twoWordsOccurrences = textFile.filter(line => line.contains("patient"))
                                  .filter(line => line.contains("admitted"))
twoWordsOccurrences.count()

(1) Spark Jobs

twoWordsOccurrences: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[12] at filter at comman
d-271712840072381:1
res3: Long = 7

Command took 2.19 seconds -- by hahme107@uottawa.ca at 11/1/2021, 8:18:23 PM on cluster_assignment_2
```

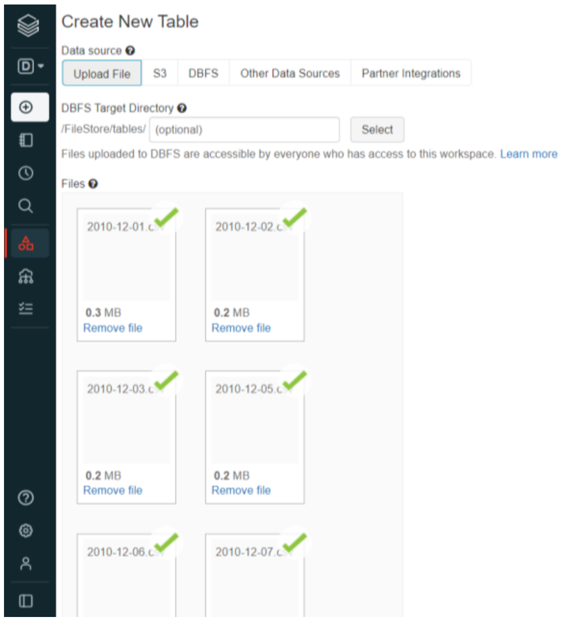
Show the output of the the word "patient" and "admitted" on the same line occurrences



```
Cmd 6

/* showing all lines contains the word "patient"
and "admitted" on the same line of text */

val twoWordsDF = twoWordsOccurrences.toDF()
twoWordsDF.show(false)

(4) Spark Jobs

+---------------------------------------------------------------------------------------------------------------------------------+
|value                                                                                                                            |
+---------------------------------------------------------------------------------------------------------------------------------+
|Your patient was admitted under the care of   Swenk , Danyel A  with a preliminary diagnosis of L HIP FX .                      |
|The patient 's disposition at the end of the visit was admitted as an inpatient to  West Texas Va Health Care System   .         |
|The patient was admitted for the lap coli with an intraoperative cholangiogram by Dr.    Ellenburg   under general anesthesia .  |
|The patient was admitted to floor .                                                                                              |
|The patient was admitted to the Medical Service for management of her congestive heart failure .                                |
|DISCHARGE DATE : The patient was admitted to the hospital on  July 15th  for a chole with a principle diagnosis of recurrent biliary colic .  |
|The patient was admitted in  December  of 1999 , at that time with anasarca and congestive heart failure , responsive to diuretics and ACE inhibitors .|
+---------------------------------------------------------------------------------------------------------------------------------+

Command took 2.70 seconds -- by hahme107@uottawa.ca at 11/1/2021, 8:20:27 PM on cluster_assignment_2
```

2. Retail Data Analysis

a) Uploaded the files to DBFS table space


Create New Table

Data source

Upload File | S3 | DBFS | Other Data Sources | Partner Integrations

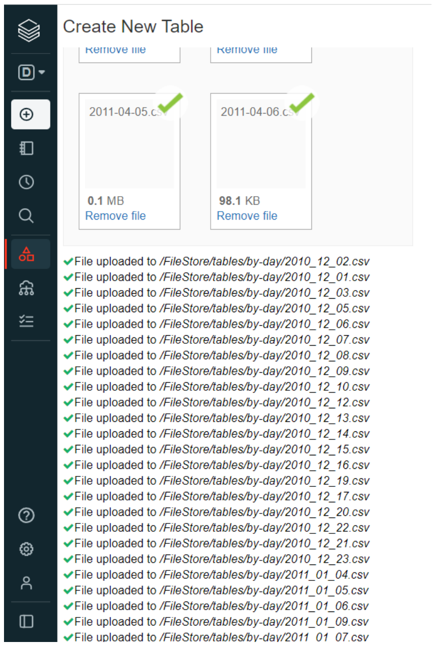
DBFS Target Directory

/FileStore/tables/ (optional) Select

Files uploaded to DBFS are accessible by everyone who has access to this workspace. Learn more

Files

2010-12-01.c 0.3 MB Remove file	2010-12-02.c 0.2 MB Remove file
2010-12-03.c 0.2 MB Remove file	2010-12-05.c 0.2 MB Remove file
2010-12-06.c	2010-12-07.c


Create New Table

2011-04-05.c 0.1 MB Remove file	2011-04-06.c 98.1 KB Remove file

```
✓File uploaded to /FileStore/tables/by-day/2010_12_02.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_01.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_03.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_05.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_06.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_07.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_08.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_09.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_10.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_12.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_13.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_14.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_15.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_16.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_19.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_17.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_20.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_22.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_21.csv
✓File uploaded to /FileStore/tables/by-day/2010_12_23.csv
✓File uploaded to /FileStore/tables/by-day/2011_01_04.csv
✓File uploaded to /FileStore/tables/by-day/2011_01_05.csv
✓File uploaded to /FileStore/tables/by-day/2011_01_06.csv
✓File uploaded to /FileStore/tables/by-day/2011_01_09.csv
✓File uploaded to /FileStore/tables/by-day/2011_01_07.csv
```

b) Total number of rows= 541909

```
#b number of rows
data.count()
```

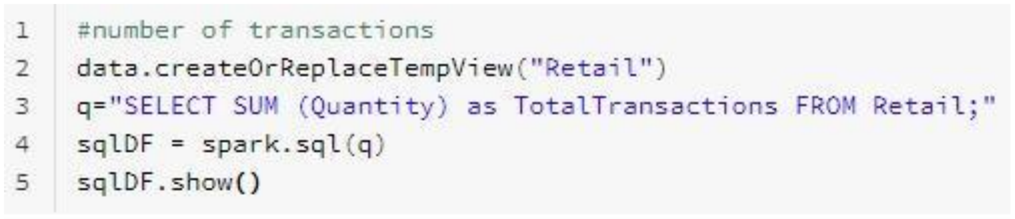
```
Out[28]: 541909
```

Total number of unique InvoiceNo= 25900

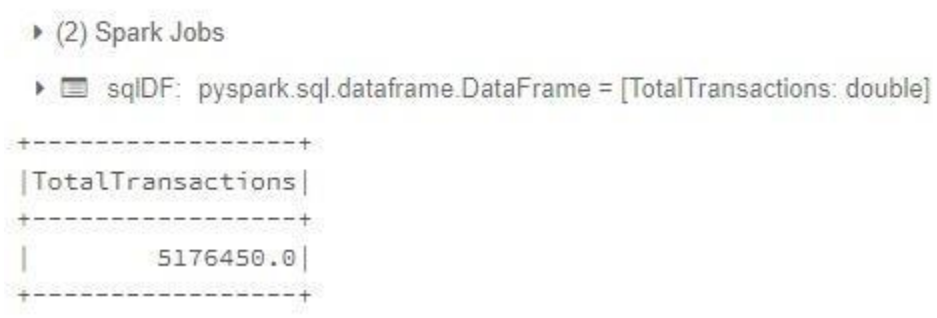
```
#number of distinct InvoiceNo
data.createOrReplaceTempView("Retail")
q="SELECT DISTINCT InvoiceNo FROM Retail;"
sqlDF = spark.sql(q)
sqlDF.count()
```

```
Out[29]: 25900
```

Total Number of transactions = Sum(Quantity)= 5176450



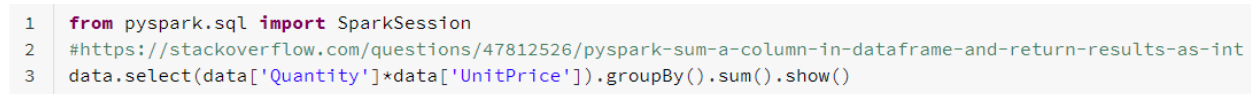
```
#number of transactions
data.createOrReplaceTempView("Retail")
q="SELECT SUM (Quantity) as TotalTransactions FROM Retail;"
sqlDF = spark.sql(q)
sqlDF.show()
```



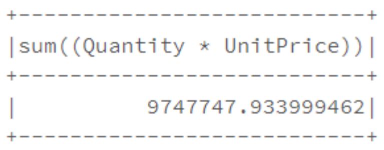
```
▸ (2) Spark Jobs
▸ sqlDF:  pyspark.sql.dataframe.DataFrame = [TotalTransactions: double]
+-----------------+
|TotalTransactions|
+-----------------+
|        5176450.0|
+-----------------+
```

The total value of the transactions = sum((Quantity * UnitPrice)) = 9747747.9

Note: the number may be slightly changed due to the floating point problem in python.

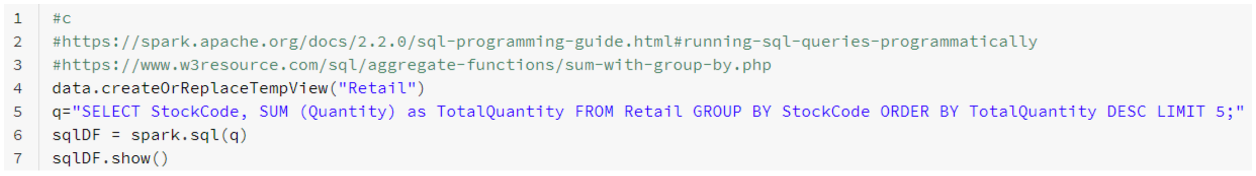


```
from pyspark.sql import SparkSession
#https://stackoverflow.com/questions/47812526/pyspark-sum-a-column-in-dataframe-and-return-results-as-int
data.select(data['Quantity']*data['UnitPrice']).groupBy().sum().show()
```



```
+--------------------------+
|sum((Quantity * UnitPrice))|
+--------------------------+
|         9747747.933999462|
+--------------------------+
```

c) The 5 top-selling products = Sum of ‘Quantity’ for each unique ‘StockCode’.



```
#c
#https://spark.apache.org/docs/2.2.0/sql-programming-guide.html#running-sql-queries-programmatically
#https://www.w3resource.com/sql/aggregate-functions/sum-with-group-by.php
data.createOrReplaceTempView("Retail")
q="SELECT StockCode, SUM (Quantity) as TotalQuantity FROM Retail GROUP BY StockCode ORDER BY TotalQuantity DESC LIMIT 5;"
sqlDF = spark.sql(q)
sqlDF.show()
```

```
+---------+-------------+
|StockCode|TotalQuantity|
+---------+-------------+
|    22197|      56450.0|
|    84077|      53847.0|
|   85099B|      47363.0|
|   85123A|      38830.0|
|    84879|      36221.0|
+---------+-------------+
```

d) The 5 top most valuable products = Sum of ‘Quantity’ * ‘UnitPrice’ for each unique ‘StockCode’.

Note: cast(number as decimal(10,2)) will take the first **two** digits after the decimal number to avoid the fractions made by the floating point.

```
#d
data.createOrReplaceTempView("Retail")
q="SELECT StockCode, cast(SUM (Quantity*UnitPrice) as decimal(10,2)) AS TotalPrice FROM Retail GROUP BY StockCode ORDER BY TotalPrice DESC LIMIT 5;"
sqlDF = spark.sql(q)
sqlDF.show()
```

```
+---------+----------+
|StockCode|TotalPrice|
+---------+----------+
|      DOT| 206245.48|
|    22423| 164762.19|
|    47566|  98302.98|
|   85123A|  97894.50|
|   85099B|  92356.03|
+---------+----------+
```

e) Each country and the total value purchases = Sum of ‘Quantity’ * ‘UnitPrice’ for each unique ‘Country’

```
#e
data.createOrReplaceTempView("Retail")
q="SELECT Country, cast(SUM (Quantity*UnitPrice) as decimal(10,2)) AS TotalValue FROM Retail GROUP BY Country;"
sqlDF = spark.sql(q)
sqlDF.show()
```

```
+---------------+----------+
|        Country|TotalValue|
+---------------+----------+
|         Sweden|  36595.91|
|        Germany| 221698.21|
|         France| 197403.90|
|         Greece|   4710.52|
|        Belgium|  40910.96|
|        Finland|  22326.74|
|          Malta|   2505.47|
|    Unspecified|   4749.79|
|          Italy|  16890.51|
|           EIRE| 263276.82|
|         Norway|  35163.46|
|          Spain|  54774.58|
|        Denmark|  18768.14|
|      Hong Kong|  10117.04|
|        Iceland|   4310.00|
|Channel Islands|  20086.29|
|            USA|   1730.92|
|    Switzerland|  56385.35|
```

And we can order the countries to see the most performing countries in purchases

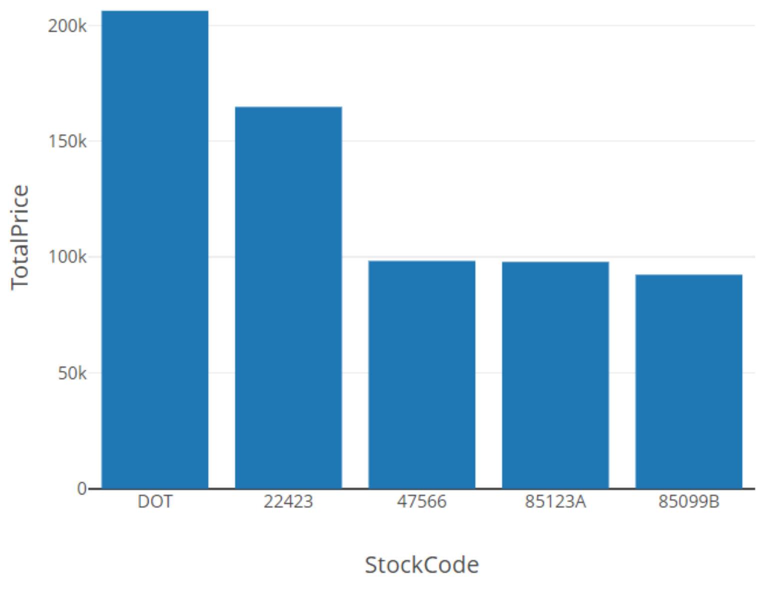
```
#In ordered way
data.createOrReplaceTempView("Retail")
q="SELECT Country, cast(SUM (Quantity*UnitPrice) as decimal(10,2)) AS TotalValue FROM Retail GROUP BY Country ORDER BY TotalValue DESC;"
sqlDF = spark.sql(q)
sqlDF.show()
```

```
+---------------+----------+
|        Country|TotalValue|
+---------------+----------+
| United Kingdom|8187806.36|
|    Netherlands| 284661.54|
|           EIRE| 263276.82|
|        Germany| 221698.21|
|         France| 197403.90|
|      Australia| 137077.27|
|    Switzerland|  56385.35|
|          Spain|  54774.58|
|        Belgium|  40910.96|
|         Sweden|  36595.91|
|          Japan|  35340.62|
|         Norway|  35163.46|
|       Portugal|  29367.02|
|        Finland|  22326.74|
|Channel Islands|  20086.29|
|        Denmark|  18768.14|
|          Italy|  16890.51|
|         Cyprus|  12946.29|
```

f) The graphical representation of ‘d’: We will use the bar chart that will be revealed when we use the display function.

```
#f
data.createOrReplaceTempView("Retail")
q="SELECT StockCode, cast(SUM (Quantity*UnitPrice) as decimal(10,2)) AS TotalPrice FROM Retail GROUP BY StockCode ORDER BY TotalPrice DESC LIMIT 5;"
sqlDF = spark.sql(q)
```

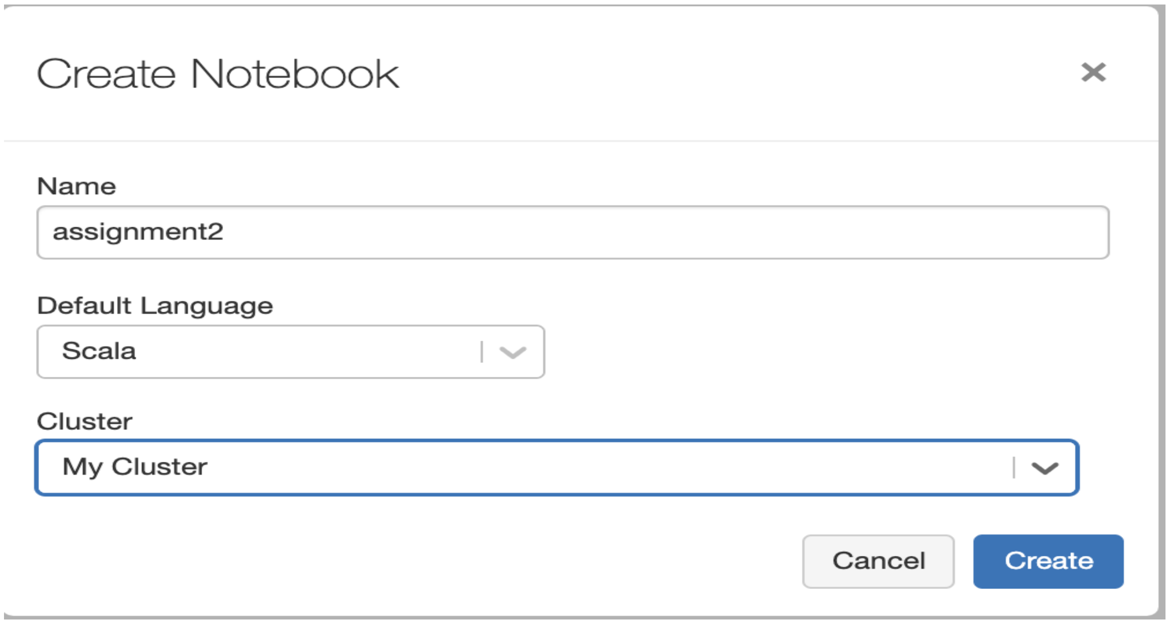
```
display(sqlDF)
```



As we can see ‘DOT’ StockCode is remarkably the most valuable product.

3. Structure Streaming

a) Create a new notebook.


Create Notebook

Name: assignment2

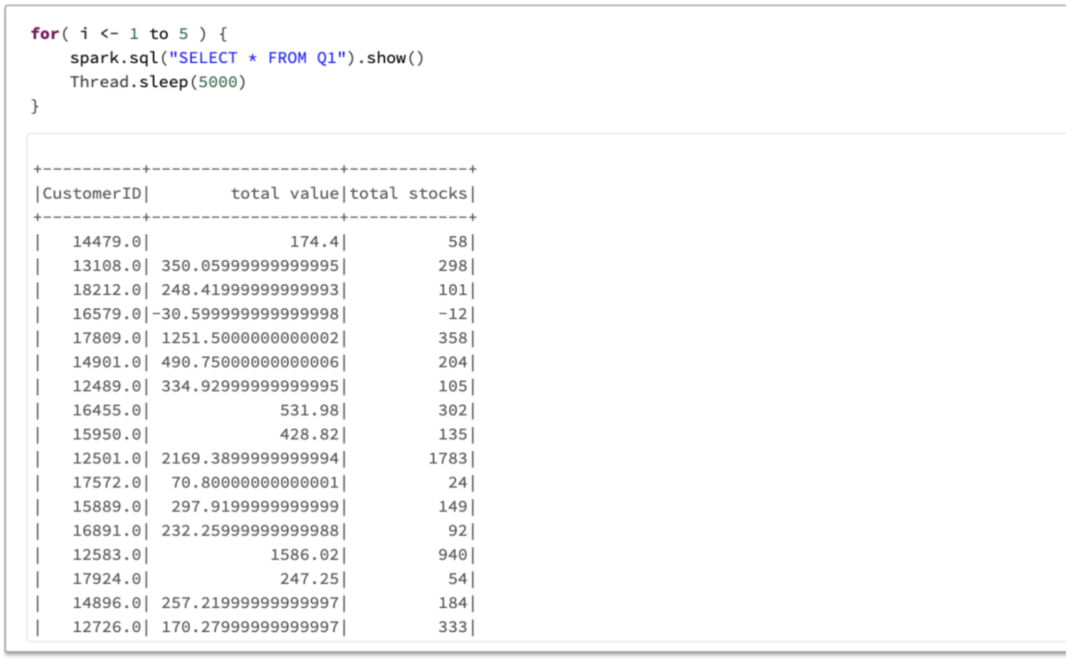
Default Language: Scala

Cluster: My Cluster

b) Load the retail data as a stream, at 20 files per trigger. For each batch pulled, capture the customer stock aggregates – total stocks, total value

```
/* Setting up a schema that reads 20 file per trigger */
val streaming = spark
                  .readStream.schema(dataSchema)
                  .option("maxFilesPerTrigger", 20)
                  .csv("/FileStore/tables/ass2")

streaming: org.apache.spark.sql.DataFrame = [InvoiceNo: string, StockCode: string ... 6 more fields]
```

c) For each batch of the input stream, create a new stream that populates another dataframe or dataset with progress for each loaded set of data. This data set should have the columns – TriggerTime (Date/Time), Records Imported, Sale value (Total value of transactions)

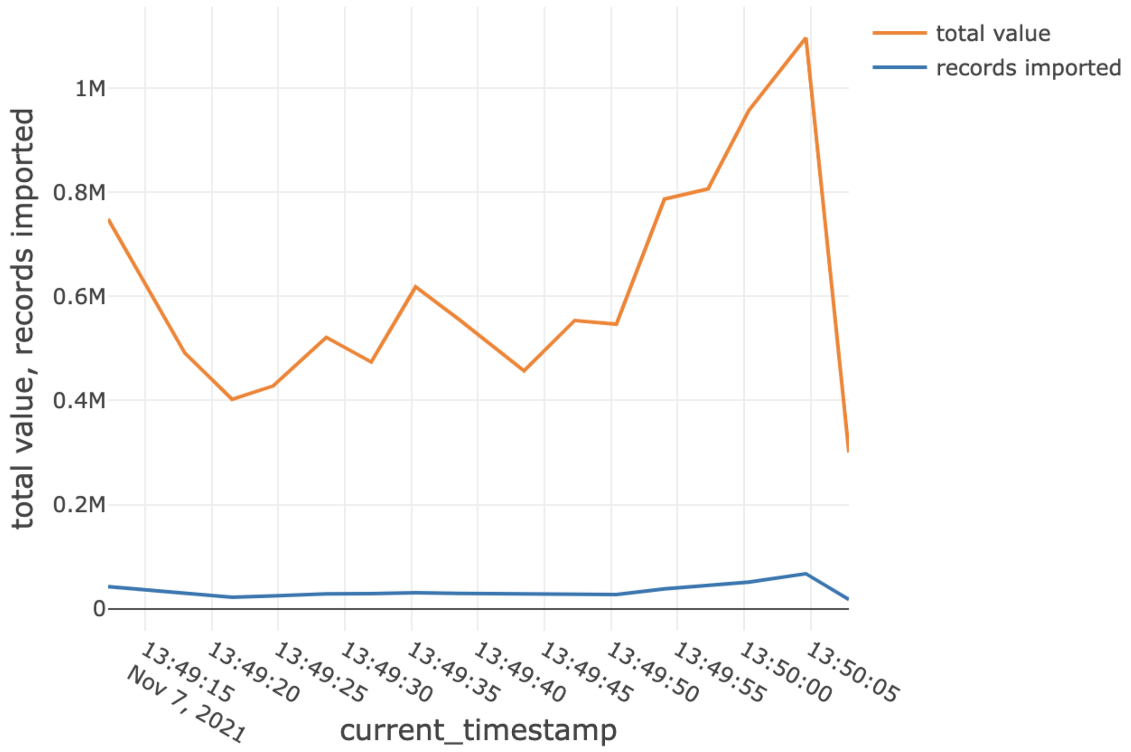


```
for( i <- 1 to 5 ) {
    spark.sql("SELECT * FROM Q1").show()
    Thread.sleep(5000)
}
```

```
+----------+-------------------+------------+
|CustomerID|        total value|total stocks|
+----------+-------------------+------------+
|   14479.0|              174.4|          58|
|   13108.0| 350.05999999999995|         298|
|   18212.0| 248.41999999999993|         101|
|   16579.0|-30.599999999999998|         -12|
|   17809.0| 1251.5000000000002|         358|
|   14901.0| 490.75000000000006|         204|
|   12489.0| 334.92999999999995|         105|
|   16455.0|             531.98|         302|
|   15950.0|             428.82|         135|
|   12501.0| 2169.3899999999994|        1783|
|   17572.0|  70.80000000000001|          24|
|   15889.0|  297.9199999999999|         149|
|   16891.0| 232.25999999999988|          92|
|   12583.0|            1586.02|         940|
|   17924.0|             247.25|          54|
|   14896.0| 257.21999999999997|         184|
|   12726.0| 170.27999999999997|         333|
```

d) Use the dataset from step (c) to plot a line graph of the import process – showing two timelines – records imported and sale values.

```
display(df.select("current_timestamp","total value","records imported"))
```


total value
records imported
1M
0.8M
0.6M
0.4M
0.2M
total value, records imported
13:49:15
Nov 7, 2021
13:49:20
13:49:25
13:49:30
13:49:35
13:49:40
13:49:45
13:49:50
13:49:55
13:50:00
13:50:05
current_timestamp