**Sample Demo :**

**ETL Pipeline in Apache Spark Within ease Python**

**Objectives:**

* Create simple but robust ETL pipelines
* Learn how Spark provides APIs to transform different data format into Data frames and SQL for analysis purpose
* How one data source could be transformed into another without any hassle

**What is Apache Spark?**

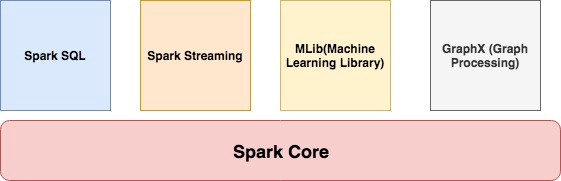
*Apache Spark is an open-source distributed general-purpose cluster-computing framework. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.*

*Apache Spark™ is a unified analytics engine for large-scale data processing.*

In short, Apache Spark is a framework which is used for processing, querying and analyzing Big data. Since the computation is done in memory hence it’s multiple fold fasters than the competitors like MapReduce and others. The rate at which terabytes of data is being produced every day, there was a need for a solution that could provide real-time analysis at high speed. Some of the Spark features are:

* It is 100 times faster than traditional large-scale data processing frameworks.
* Easy to use as you can write Spark applications in Python, R, and Scala.
* It provides libraries for SQL, Steaming and Graph computations.

**Apache Spark Components**



**Spark Core**

It contains the basic functionality of Spark like task scheduling, memory management, interaction with storage, etc.

**Spark SQL**

It is a set of libraries used to interact with structured data. It used an SQL like interface to interact with data of various formats like CSV, JSON, Parquet, etc.

**Spark Streaming**

Spark Streaming is a Spark component that enables the processing of live streams of data. Live streams like Stock data, Weather data, Logs, and various others.

**MLib**

MLib is a set of Machine Learning Algorithms offered by Spark for both supervised and unsupervised learning

**GraphX**

It is Apache Spark’s API for graphs and graph-parallel computation. It extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge. It provides a uniform tool for ETL, exploratory analysis and iterative graph computations.

**Spark Cluster Managers**

Spark supports the following resource/cluster managers:

* **Spark Standalone** — a simple cluster manager included with Spark
* **Apache Mesos** — a general cluster manager that can also run Hadoop applications.
* **Apache Hadoop YARN** — the resource manager in Hadoop 2
* **Kubernetes** — an open source system for automating deployment, scaling, and management of containerized applications.

**Setup and Installation**

<https://spark.apache.org/downloads.html>

Download the binary of Apache Spark from .

You must have Scala installed on the system and its path should also be set.

For this tutorial, we are using version 2.4.3 which was released in May 2019. Move the folder in /usr/local

mv spark-2.4.3-bin-hadoop2.7 /usr/local/spark

And then export the path of both Scala and Spark.

#Scala Path  
export PATH="/usr/local/scala/bin:$PATH"#Apache Spark path  
export PATH="/usr/local/spark/bin:$PATH"

Invoke the Spark Shell by running the spark-shell command on your terminal. If all goes well, you will see something like below:



It loads the Scala based shell. Since we are going to use Python language then we have to install **PySpark**.

pip install pyspark

Once it is installed you can invoke it by running the command pyspark in your terminal:



You find a typical Python shell but this is loaded with Spark libraries.

**Development in Python**

Let’s start writing our first program.

from pyspark.sql import SparkSession  
from pyspark.sql import SQLContextif \_\_name\_\_ == '\_\_main\_\_':  
 scSpark = SparkSession \  
 .builder \  
 .appName("reading csv") \  
 .getOrCreate()

We have imported two libraries: SparkSession and SQLContext.

SparkSession is the entry point for programming Spark applications. It let you interact with DataSet and DataFrame APIs provided by Spark. We set the application name by calling appName. The getOrCreate() method either returns a new SparkSession of the app or returns the existing one.

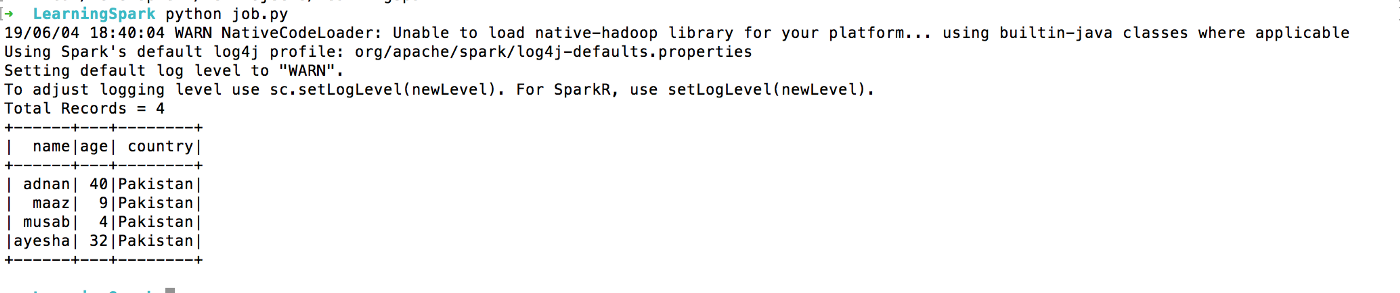
Our next objective is to read CSV files. I have created a sample CSV file, called data.csv which looks like below:

name,age,country  
adnan,40,Pakistan  
maaz,9,Pakistan  
musab,4,Pakistan  
ayesha,32,Pakistan

And the code:

if \_\_name\_\_ == '\_\_main\_\_':  
 scSpark = SparkSession \  
 .builder \  
 .appName("reading csv") \  
 .getOrCreate()data\_file = '/Development/PetProjects/LearningSpark/data.csv'  
 sdfData = scSpark.read.csv(data\_file, header=True, sep=",").cache()  
 print('Total Records = {}'.format(sdfData.count()))  
 sdfData.show()

I set the file path and then called .read.csv to read the CSV file. The parameters are self-explanatory. The .cache() caches the returned resultset hence increase the performance. When I run the program it returns something like below:



Read multiple files in a dataframe. Let’s create another file, I call it data1.csv and it looks like below:

1

2

3

4

5

name,age,country

noreen,23,England

Aamir,9,Pakistan

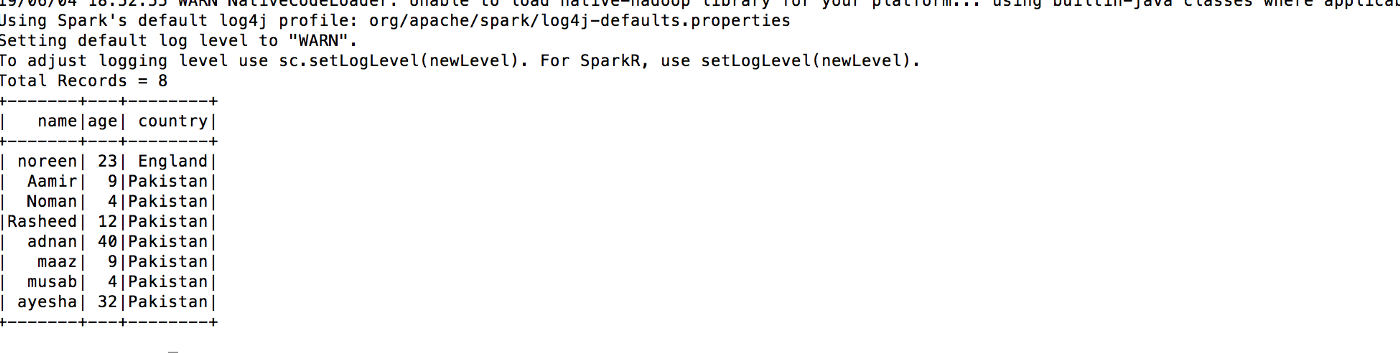
Noman,4,Pakistan

Rasheed,12,Pakistan

All I have to do this:

data\_file = '/Development/PetProjects/LearningSpark/data\*.csv' and it will read all files starts with *data*and of type CSV.

What it will do that it’d read all CSV files that match a pattern and dump result:



Dumps all the data from the CSVs into a single dataframe. Pretty cool huh.

But one thing, this dumping will only work if all the CSVs follow a certain schema. If you have a CSV with different column names then it’s gonna return the following message.

19/06/04 18:59:05 WARN CSVDataSource: Number of column in CSV header is not equal to number of fields in the schema:  
 Header length: 3, schema size: 17  
CSV file: file:///Development/PetProjects/LearningSpark/data.csv

Spark complains about CSV files that are not the same are unable to be processed.

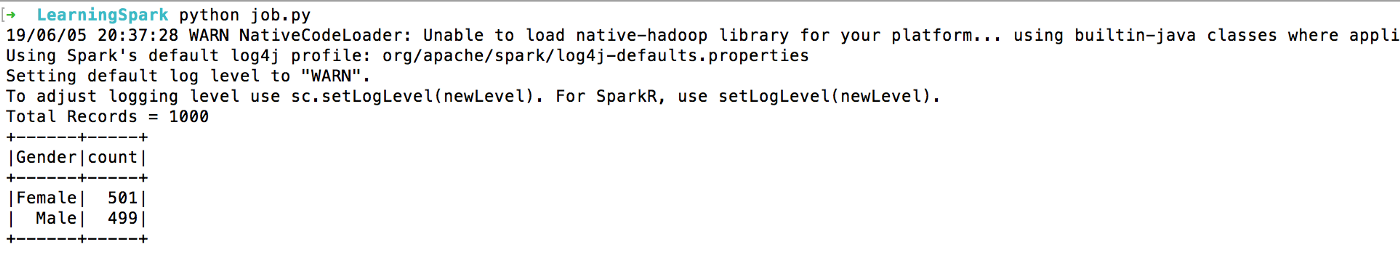
You can perform many operations with DataFrame but Spark provides you much easier and familiar interface to manipulate the data by using SQLContext. It is the gateway to SparkSQL which lets you use SQL like queries to get the desired results.

Before we move further, let’s play with some real data. For that purpose, we are using Supermarket’s sales data which I got from [Kaggle](https://www.kaggle.com/aungpyaeap/supermarket-sales).

Before we try SQL queries, let’s try to group records by Gender. We are dealing with the **EXTRACT** part of the ETL here.

data\_file = '/Development/PetProjects/LearningSpark/supermarket\_sales.csv'  
sdfData = scSpark.read.csv(data\_file, header=True, sep=",").cache()gender = sdfData.groupBy('Gender').count()  
print(gender.show())

When you run, it returns something like below:

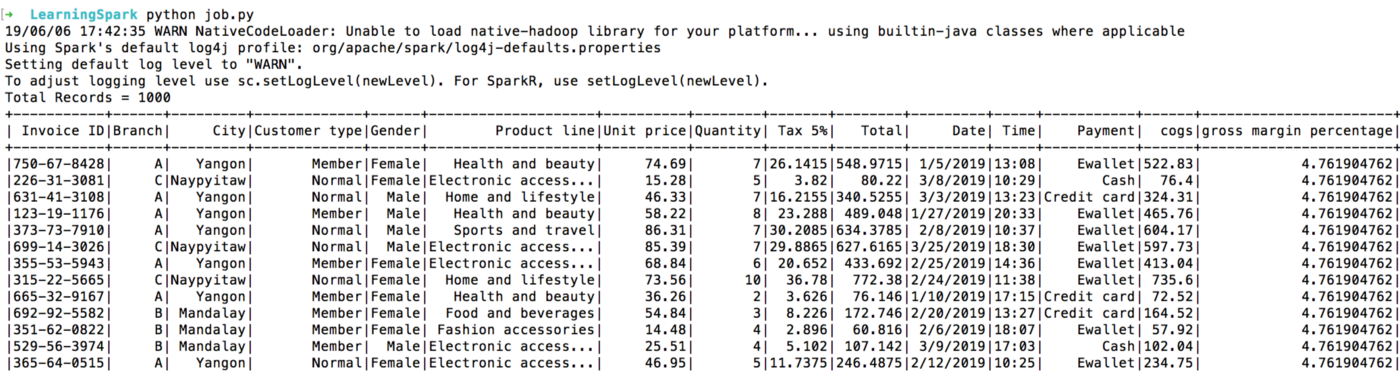


groupBy() groups the data by the given column. In our case, it is the **Gender**column.

SparkSQL allows you to use SQL like queries to access the data.

sdfData.registerTempTable("sales")  
output = scSpark.sql('SELECT \* from sales')  
output.show()

First, we create a temporary table out of the dataframe. For that purpose registerTampTable is used. In our case the table name is **sales**. Once it’s done you can use typical SQL queries on it. In our case it is **Select \* from sales**.

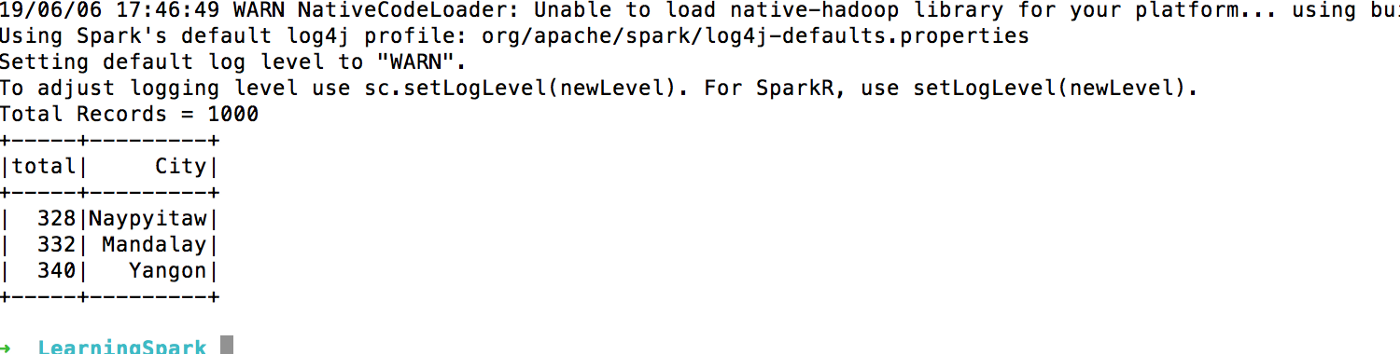


Or something like below:

output = scSpark.sql('SELECT \* from sales WHERE `Unit Price` < 15 AND Quantity < 10')  
output.show()

Or even aggregated values.

output = scSpark.sql('SELECT COUNT(\*) as total, City from sales GROUP BY City')  
output.show()

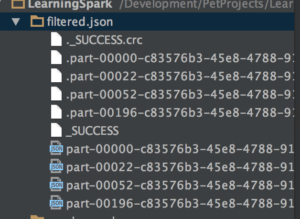


We are just done with the **TRANSFORM** part of the ETL here.

Finally the **LOAD** part of the ETL. What if you want to save this transformed data? Well, you have many options available, RDBMS, XML or JSON.

output.write.format('json').save('filtered.json')

When you run it Sparks create the following folder/file structure.



It created a folder with the name of the file, in our case it is **filtered.json**. Then, a file with the name **\_SUCCESS**tells whether the operation was a success or not. In case it fails a file with the name **\_FAILURE** is generated. Then, you find multiple files here. The reason for multiple files is that each work is involved in the operation of writing in the file. If you want to create a single file(which is not recommended) then coalesce can be used that collects and reduces the data from all partitions to a single dataframe.

output.coalesce(1).write.format('json').save('filtered.json')

And it will output the following data:

{"total":328,"City":"Naypyitaw"}  
{"total":332,"City":"Mandalay"}  
{"total":340,"City":"Yangon"}

**MySQL and Apache Spark Integration**

The above dataframe contains the transformed data. We would like to load this data into MYSQL for further usage like Visualization or showing on an app.

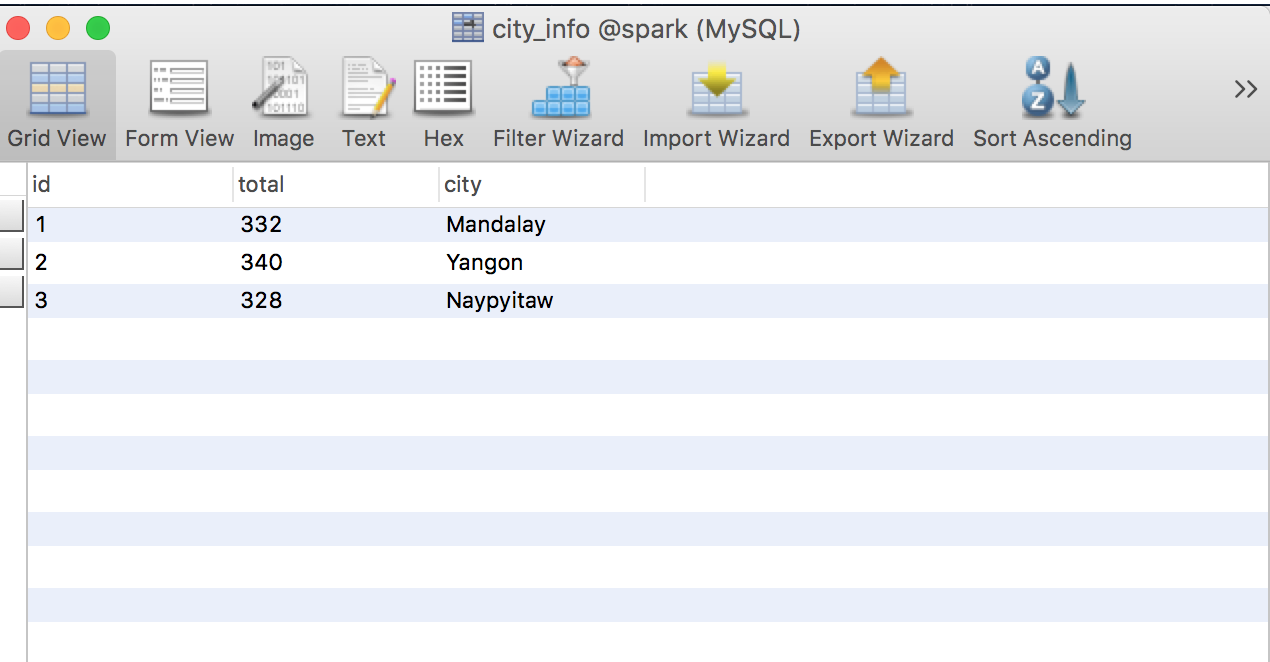
First, we need the MySQL connector library to interact with Spark. We will download the connector from [MySQL website](https://cdn.mysql.com/Downloads/Connector-J/mysql-connector-java-8.0.16.tar.gz) and put it in a folder. We will amend SparkSession to include the JAR file.

scSpark = SparkSession \  
 .builder \  
 .appName("reading csv") \  
 .config("spark.driver.extraClassPath", "/usr/local/spark/jars/mysql-connector-java-8.0.16.jar") \  
 .getOrCreate()

The output now looks like below:

output = scSpark.sql('SELECT COUNT(\*) as total, City from sales GROUP BY City')  
 output.show()  
 output.write.format('jdbc').options(  
 url='jdbc:mysql://localhost/spark',  
 driver='com.mysql.cj.jdbc.Driver',  
 dbtable='city\_info',  
 user='root',  
 password='root').mode('append').save()

I created the required Db and table in my DB before running the script. If all goes well you should see the result like below:



Spark makes it easier to transfer data from One data source to another.