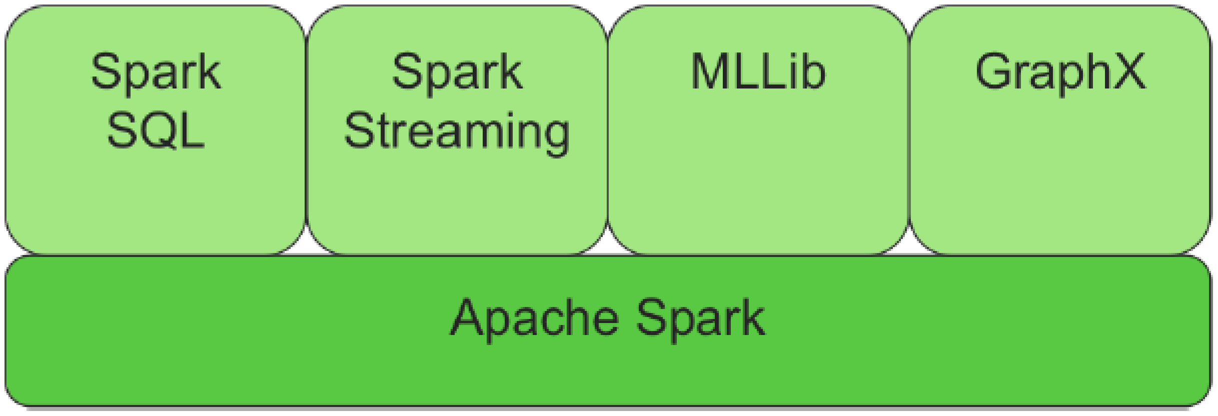
Spark

Introduction:

Spark: is open source framework. It's part of Hadoop eco-system. It's produced by Apache. It's general purpose application. It's used for data processing for regular files and HDFS files. Originally it depends on Map Reduce concept for data processing. It's uses memory during the processing.

Spark is created by Berkeley Data Analytics Stack. Spark API's are available in Scala, Python, Java and recently add R. Spark is a unified framework for data engineering and data science.

Spark RDD & Dataframe:



1- Apache Spark: Contains the Spark core API's. The main component in this part is resilient distributed datasets (RDDs). This part is a container contains the data in string format. RDD depends on key and value concept. When assign dataset to RDD, we can not change the values

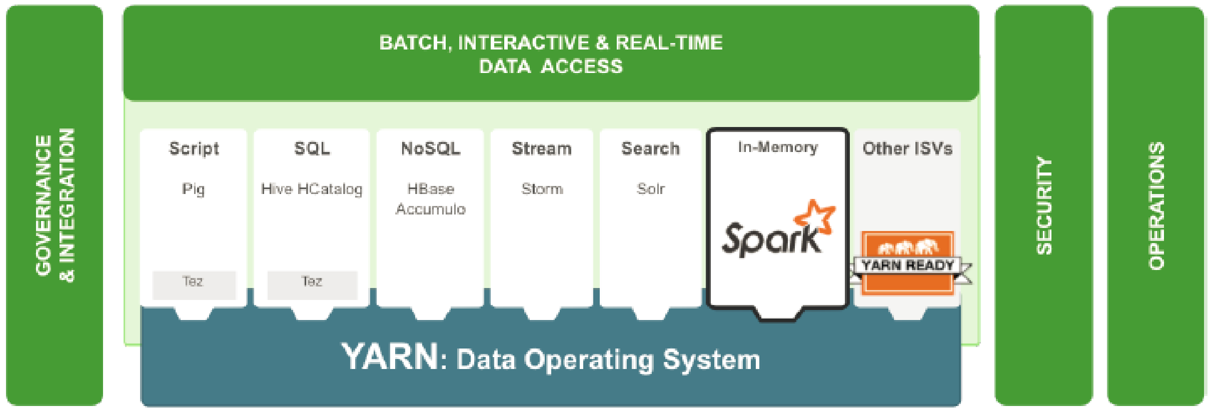
The Resilient Distributed Dataset (RDD) is an immutable collection of objects (or records) that can be
operated on in parallel. RDDs adhere to these key attributes that make up their namesake:
• Resilient: RDDS can be recreated from parent RDDs, and an RDD keeps its lineage information
• Distributed: Partitions of data are distributed across nodes in the cluster
• Dataset: A set of data than can be accessed

2- Spark SQL: It's the advanced version of RDD. Only this part can keep the data in structured format. This container known as dataframe. With Dataframe, you can use SQL commands or you can use it with SQL queries. In our journey, we will use Spark dataframe in the data analytics.

3- Spark streaming. Deals with process of data streaming

4- MLLIB/ML is deal with machine learning algorithms. It deals with supervised and supervised machine learning.

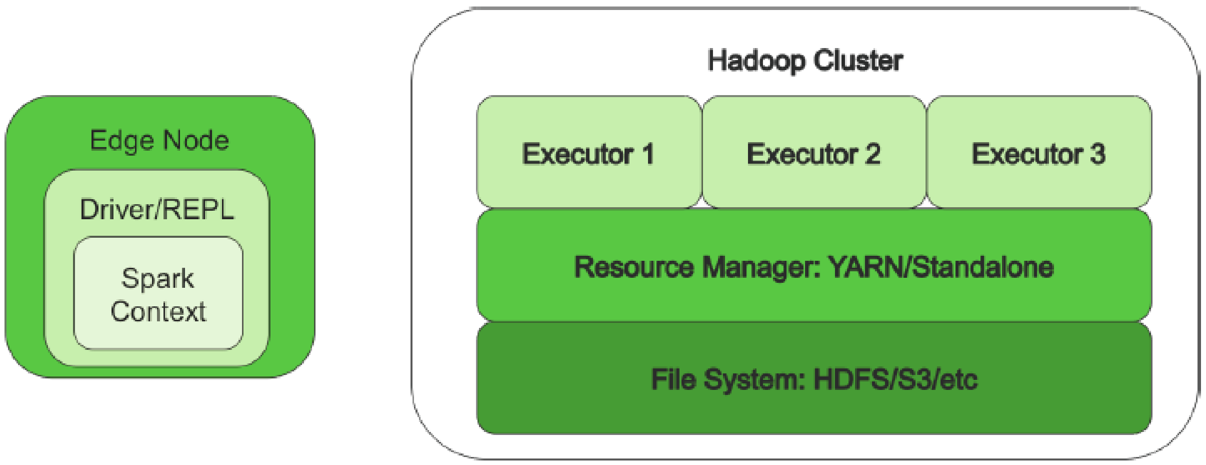
5- GraphX: deals with graph data during the process



The Hadoop framework

Spark has two main methods Transformation & Actions. We've previously mentioned that transformations don't process data until an application is call any action. This is called Lazy Evaluation. Lazy Evaluation prevents the processing of unneeded data.

Spark main parts:



Spark application aunning on a Hadoop cluster

Spark Executor: The Spark executor is the component that does almost all of the work and can be thought of as a worker. An executor is comparable to a mapper and a reducer, but not exactly the same. The executor does both map and reduce tasks, which means they do most of the work of the application

SparkContext: The SparkContext is comparable to a boss. The SparkContext contains all the code and objects required to process the data in the cluster, and works with the resource manager to get the requested resources for the application. It is also responsible for scheduling tasks following the DAG schedule. The SparkContext checks in with the executors to report work being done and provide log updates to the developer.

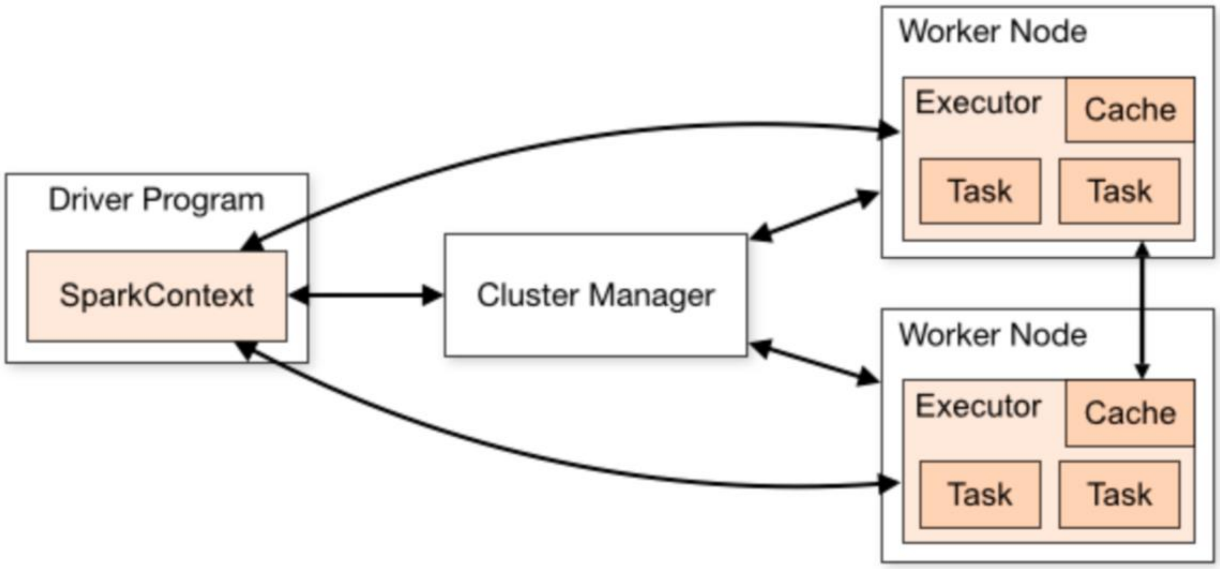
Spark Driver: The Spark driver is the owner. Spark driver is a JVM that contains the SparkContext. The driver contains the main() function and defines RDDs/Dataframe. It is allocated a predetermined amount of resources for some processing and holding the SparkContext.

Resource Manager : Spark can use two resource managers when running on a cluster. First, and most common, is YARN; second, is the Spark Standalone Resource Manager.

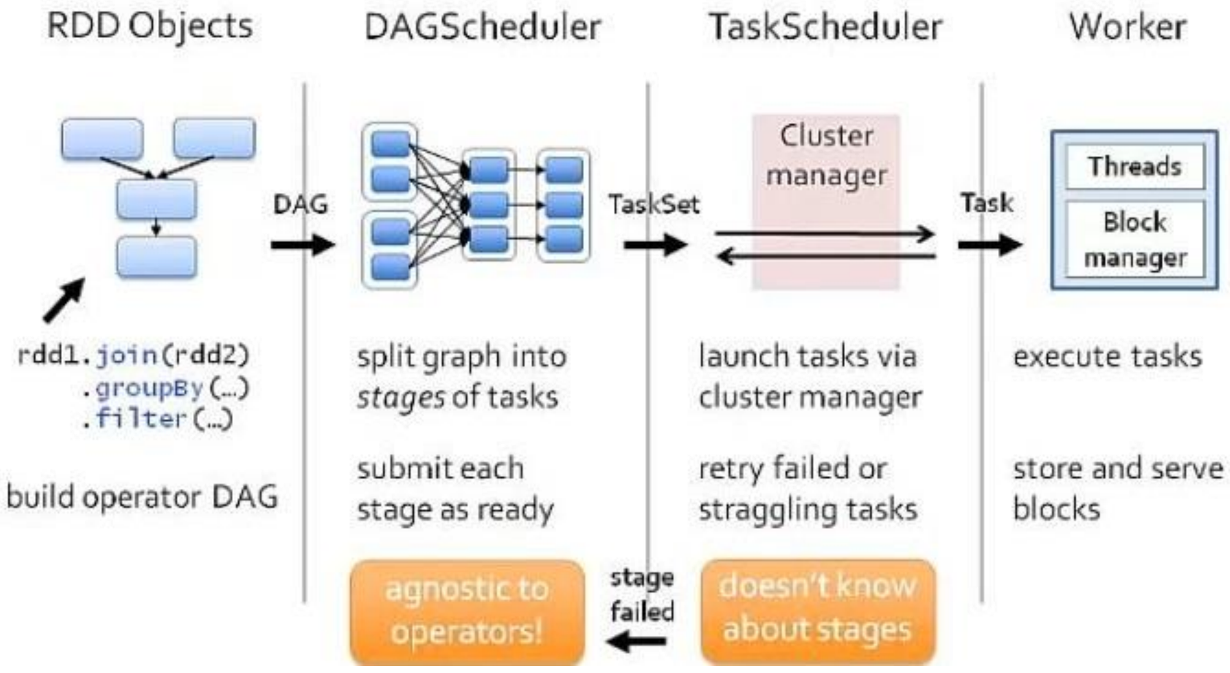
Spark start job:

1- When spark application start, creating spark context, creating DAG, breaking the job into stages and task and scheduling of the task. It defines the transformations and actions applied to the data set.
2- The driver has instantiated an object of the SparkContext class. This object allows the driver to acquire a connection to the cluster, request resources, split the application actions into tasks, and schedule and launch tasks in the executors.
3- The driver first asks the application master to allocate resources for the containers on the worker/slave nodes and create executors process.

4- Once the executors are created the driver directly coordinates with the worker nodes and assign the task to them.



When we create the RDD (After the Driver run and Spark context created). After that DAQ will create with different stages and tasks. After that the schedular task will create. The worker will start.



The main two operations of RDD are Transformations and actions.

Examples of

RDD transformation functions:

- combineByKey
- groupByKey()
- reduceByKey(func)
- mapValues(func)
- flatMapValues(func)
- keys()
- values()
- sortByKey()
- join
- cogroup
- aggregateByKey
- fullOuterJoin
- leftOuterJoin
- rightOuterJoin

Examples of actins:

RDD Action functions:

- count
- collect
- take
- top
- reduce
- foreach
- reduce

Start on Spark

There are three main methods, we can start to write Spark in python:

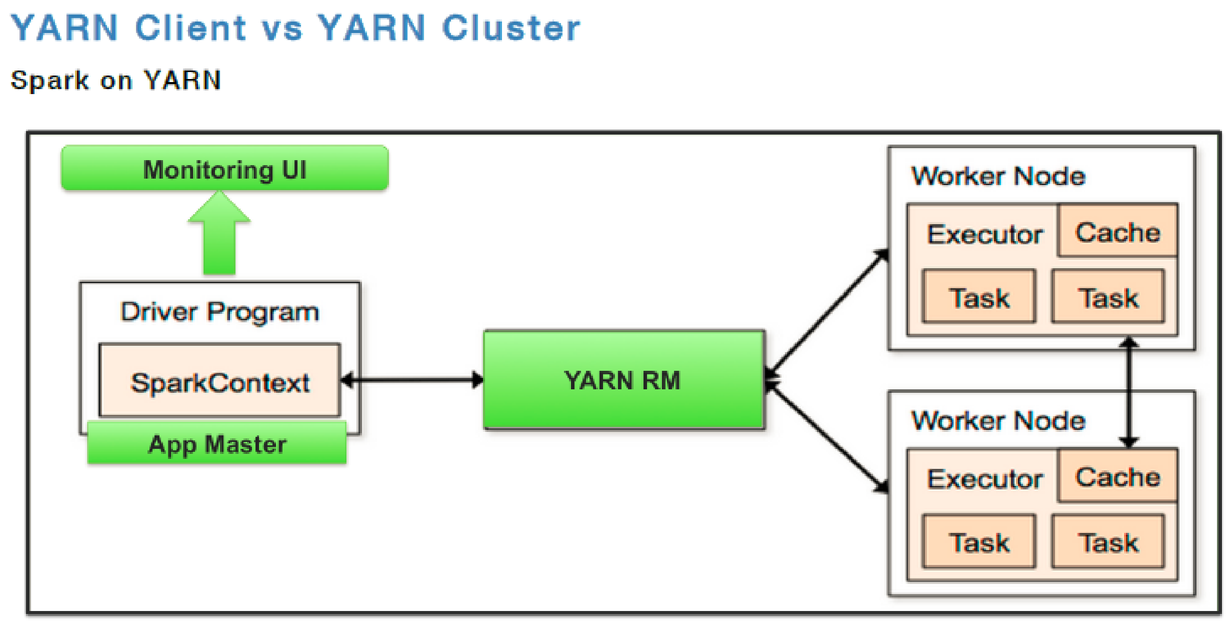
1- **Pyspark:** It's the spark interface to write spark code in python
2- **Jupyter notebook:** In Jupyter notebook we can connect to spark and run SparkSQL and Dataframe commands.
3- **Spark-submit:** After writing the code in script file, you can use spark-submit to submit the job that written in python.

spark-submit Example:

Here is a sample spark-submit example:

```
spark-submit --master yarn-client --num-executors 4 --executor-memory 8g /user/root/myApp.py
/home/root/input.parquet /home/root/output.avro
```

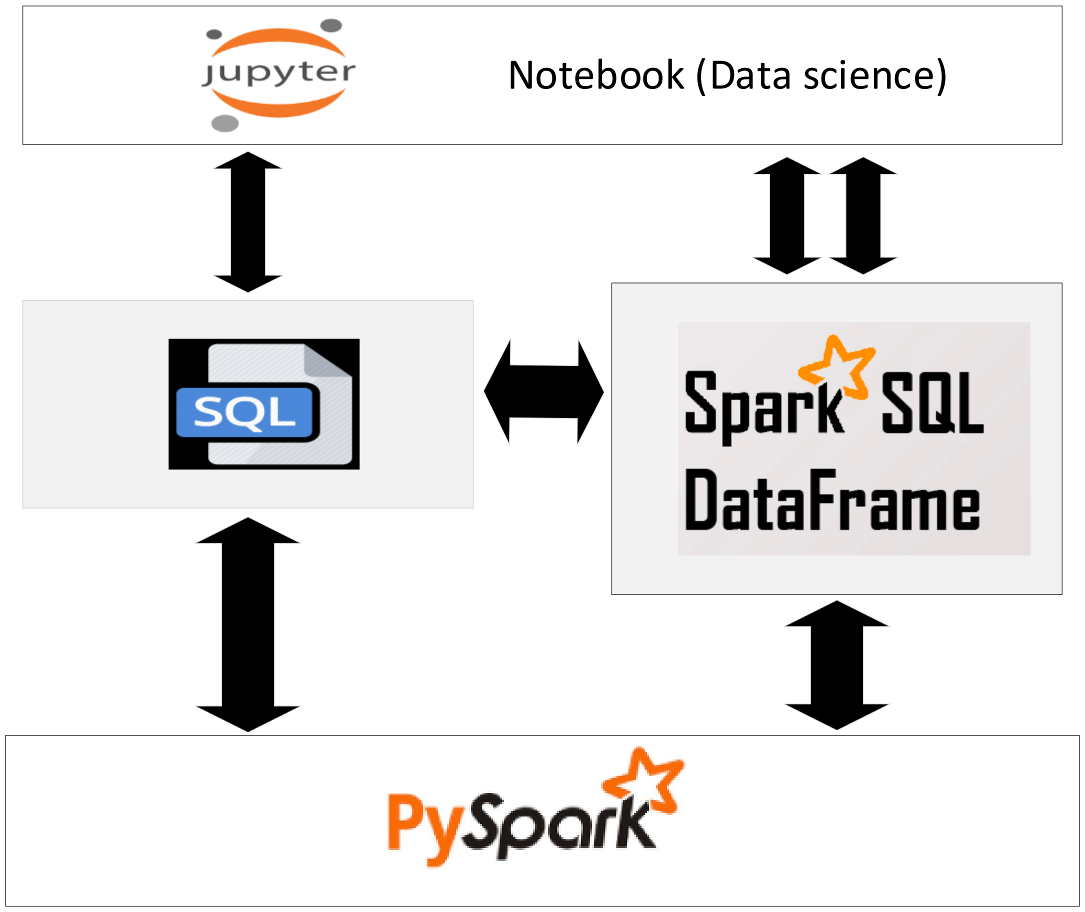
In this example, we're submitting an application to YARN in the yarn-client mode. We're requesting 4 executors, each with 8GB of memory. The application we're submitting is /user/root/myApp.py, and we're passing in two arguments.



Spark on YARN

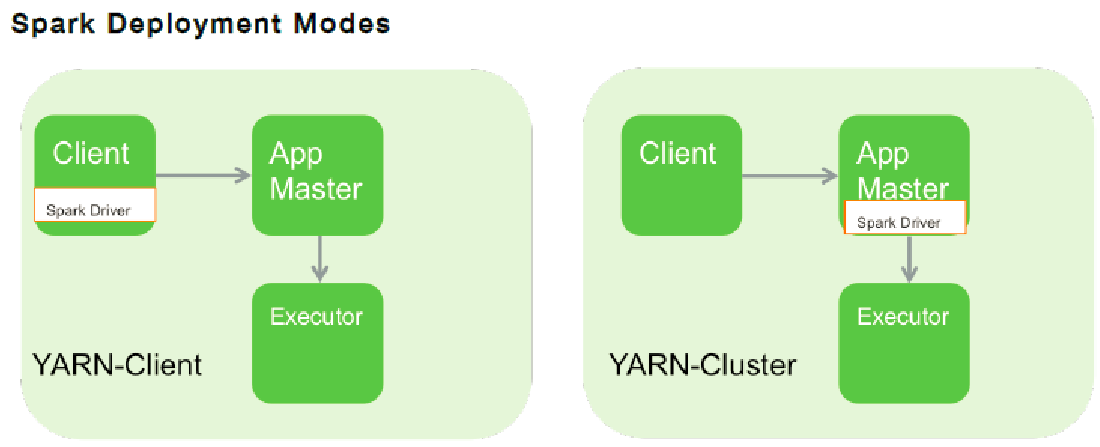
SparkSQL & Dataframe:

Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API. SparkSQL has the capabilities to use Dataframe and through Dataframe, we can run SQL. You can write a dataframe to get the data from file Linux local file, hdfs path or s3 in AWS.



YARN/REPL Application Submission:

Now that the two ways have been described on the different ways to submit (spark-submit) an application, its time for the developer to decide which submission is the best to use. Master = yarn-client & yarn-cluster



yarn-client vs yarn-cluster

In REPL (Spark-shell), there are two ways to submit the job local (local run) or yarn-client (run on cluster). We can specify it when start REPL

Job submit in Spark:

When we use REPL (pyspark) or spark-submit, we can specify some parameters to improve the execution like number of executors, number of core per executors and the memory for each executor

`**--num-executors**` = `In this approach, we'll assign one executor per node`
= `total-nodes-in-cluster`
= 10

`**--executor-cores**` = `one executor per node means all the cores of the node are assigned to one executor`

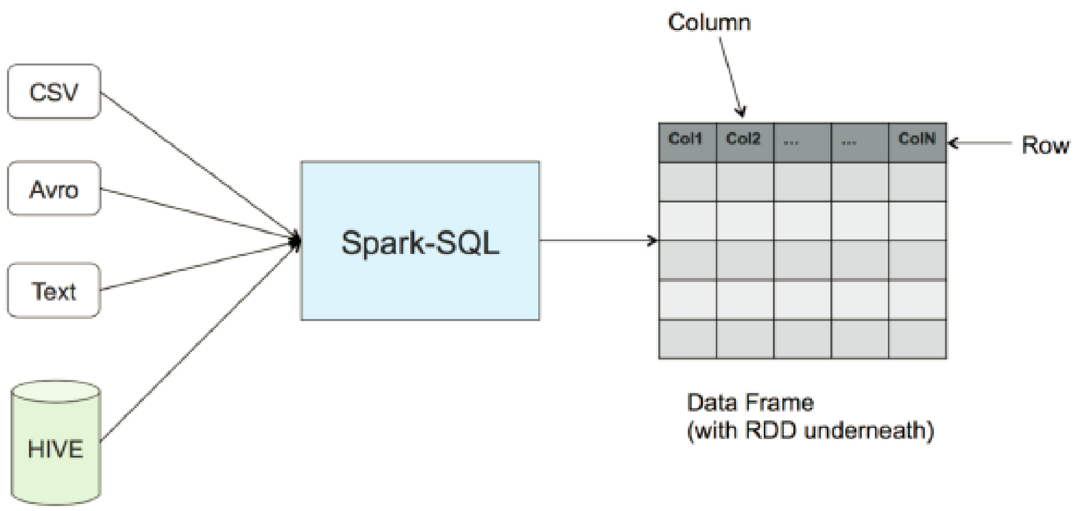
= `total-cores-in-a-node`
= 16

`**--executor-memory**` = `amount of memory per executor`

= `mem-per-node/num-executors-per-node`
= 64GB/1 = 64GB

The DataFrame Abstraction:

The DataFrame concept in Spark came from the DataFrame (dplr) concept in R, or pandas in Python. Spark SQL implemented a schema that sits on top of an RDD of row objects. This schema can than be used to interact with the data using the DataFrame API, which is available in Scala, Java, Python. Much like Hive, a DataFrame is a set of metadata that sits on top of an RDD. The RDD can be created from many file types. A DataFrame is conceptually equivalent to a table in traditional data warehousing.



the DataFrame API reader has many built-in and external plugins allowing developers to read from all sorts of sources.

Built-in include:

• Hive Tables
• JSON
• Parquet
• HDFS Text
• ORC
• JDBC

External plugins include:

• CSV
• HBase
• Avro
• elasticsearch

Programming examples with python:

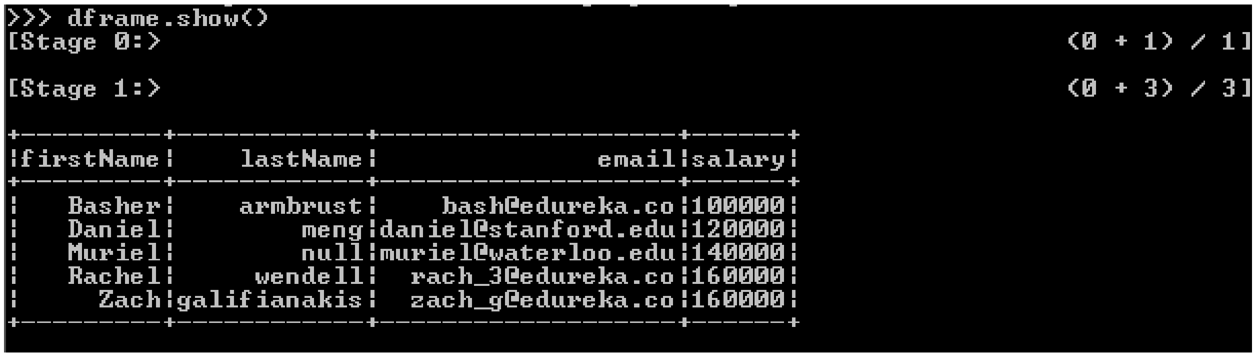
Read file from local windows or Linux:

Start pyspark:

<u>Example (1):</u> In this example, we will create employee object. Also we create 5 employee objects with data and create the data frame from these employee objects

```
>from pyspark.sql import *

>Employee = Row("firstName", "lastName", "email", "salary")

>employee1 = Employee('Basher', 'armbrust', 'bash@edureka.co', 100000)
>employee2 = Employee('Daniel', 'meng', 'daniel@stanford.edu', 120000 )
>employee3 = Employee('Muriel', None, 'muriel@waterloo.edu', 140000 )
>employee4 = Employee('Rachel', 'wendell', 'rach_3@edureka.co', 160000 )
>employee5 = Employee('Zach', 'galifianakis', 'zach_g@edureka.co', 160000 )

>employee_Seq = [employee1, employee2, employee3, employee4, employee5]
>dframe = spark.createDataFrame(employee_Seq)
>dframe.show()
```

Output:

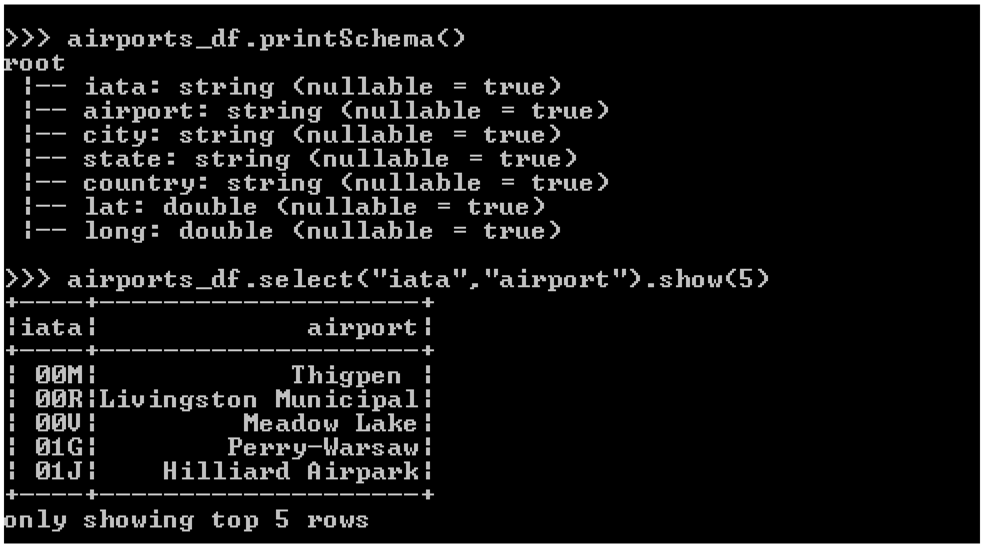


```
>>> dframe.show()
[Stage 0:>                                                          (0 + 1) / 1]

[Stage 1:>                                                          (0 + 3) / 3]

+---------+-----------+-------------------+------+
|firstName|   lastName|              email|salary|
+---------+-----------+-------------------+------+
|   Basher|   armbrust|    bash@edureka.co|100000|
|   Daniel|       meng|daniel@stanford.edu|120000|
|   Muriel|       null|muriel@waterloo.edu|140000|
|   Rachel|    wendell|  rach_3@edureka.co|160000|
|     Zach|galifianakis|  zach_g@edureka.co|160000|
+---------+-----------+-------------------+------+
```

Example (2):

Read file from c: drive with path “c:\\bigdata\\Data\\airports.csv”. and put the data in dataframe

```
airports_df = spark.read.csv("c:\\bigdata\\Data\\airports.csv", inferSchema = True, header = True)
airports_df.printSchema()
airports_df.select("iata","airport").show(5)
```



```
>>> airports_df.printSchema()
root
 |-- iata: string (nullable = true)
 |-- airport: string (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
 |-- country: string (nullable = true)
 |-- lat: double (nullable = true)
 |-- long: double (nullable = true)

>>> airports_df.select("iata","airport").show(5)
+----+--------------------+
|iata|             airport|
+----+--------------------+
| 00M|            Thigpen |
| 00R|Livingston Municipal|
| 00V|         Meadow Lake|
| 01G|        Perry-Warsaw|
| 01J|    Hilliard Airpark|
+----+--------------------+
only showing top 5 rows
```

Example(3):

Read file from hdfs drive with path “c:\\bigdata\\Data\\airports.csv”. and put the data in dataframe

(1): >start-all

(2): >hadoop fs -put airports.csv /temp/

(3): >hadoop fs -ls /temp/airports.csv

(4): >hadoop fs -ls hdfs://localhost:9000/temp/airports.csv

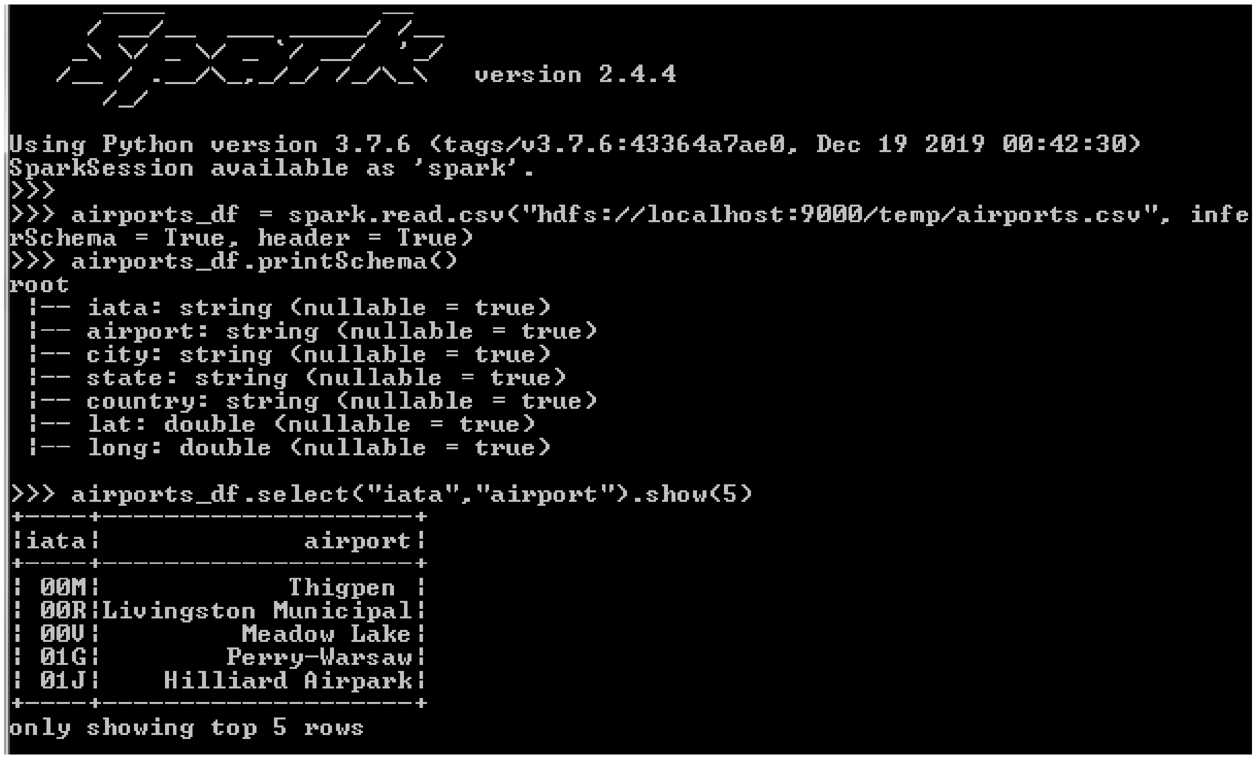
(5): >pyspark

(6): >pyspark

>airports_df = spark.read.csv("hdfs://localhost:9000/temp/airports.csv", inferSchema = True, header = True)

>airports_df.printSchema()

>airports_df.select("iata","airport").show(5)



```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.4.4
      /_/

Using Python version 3.7.6 (tags/v3.7.6:43364a7ae0, Dec 19 2019 00:42:30)
SparkSession available as 'spark'.
>>>
>>> airports_df = spark.read.csv("hdfs://localhost:9000/temp/airports.csv", infe
rSchema = True, header = True)
>>> airports_df.printSchema()
root
 |-- iata: string (nullable = true)
 |-- airport: string (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
 |-- country: string (nullable = true)
 |-- lat: double (nullable = true)
 |-- long: double (nullable = true)

>>> airports_df.select("iata","airport").show(5)
+----+--------------------+
|iata|             airport|
+----+--------------------+
| 00M|            Thigpen |
| 00R|Livingston Municipal|
| 00V|         Meadow Lake|
| 01G|        Perry-Warsaw|
| 01J|    Hilliard Airpark|
+----+--------------------+
only showing top 5 rows
```