What is PySpark ?

PySpark is nothing but the `Python API` for Apache Spark.

It offers `PySpark Shell` which connects the *Python API* to the spark core and in turn initializes the `Spark context`.

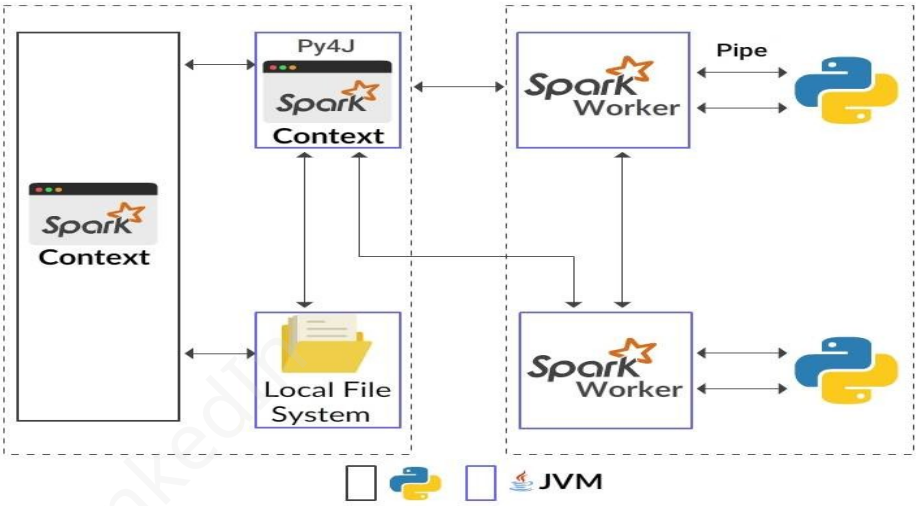
More on PySpark

- For any spark functionality, the entry point is `SparkContext`.
- SparkContext uses `Py4J` to launch a JVM and creates a `JavaSparkContext`.
- By default, PySpark has SparkContext available as `sc`, so creating a new SparkContext won't work.

Py4J

- PySpark is built on top of `Spark's Java API`.
- Data is processed in Python and `cached / shuffled` in the JVM.
- Py4J enables Python programs running in a `Python interpreter` to dynamically access `Java objects in a Java Virtual Machine`.
- Here methods are called as if the Java objects resided in the `Python interpreter and Java collections`. can be accessed through standard `Python collection methods`.

More on Py4J



- In the Python driver program, `SparkContext` uses `Py4J` to launch a `JVM` and create a `JavaSparkContext`.

- To establish local communication between the `Python and Java SparkContext objects` Py4J is used on the driver.

Installing and Configuring PySpark

- PySpark requires `Python 2.6 or higher`.
- PySpark applications are executed using a standard `CPython` interpreter in order to support `Python modules that use C extensions`.
- By default, PySpark requires python to be available on the `system PATH` and use it to run programs.
- Among PySpark's library dependencies all of them are bundled with `PySpark` including Py4J and they are `automatically imported`.

Getting Started

We can enter the `Spark's python environment` by running the given command in the shell.

```
./bin/pyspark
```

This will start your`PySpark shell`.`

```
Python 2.7.12 (default, Nov 20 2017, 18:23:56)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/
Using Python version 2.7.12 (default, Nov 20 2017 18:23:56)
SparkSession available as 'spark'.
<<<
```

Resilient Distributed Datasets (RDDs)

- `Resilient distributed datasets (RDDs)` are known as the main abstraction in Spark.
- It is a partitioned collection of objects spread across a cluster, and can be persisted in memory or on disk.
- Once RDDs are created they are immutable.

There are two ways to create RDDs:

1. Parallelizing a collection in `driver program`.
2. Referencing one dataset in an external storage system, like a shared `filesystem`, `HBase`, `HDFS`, or any data source providing a `Hadoop InputFormat`.

Features Of RDDs

- **Resilient**, i.e. tolerant to faults using `RDD lineage graph` and therefore ready to recompute `damaged` or `missing` partitions due to `node failures`.
- **Dataset** - A set of partitioned data with primitive values or values of values, For example, `records or tuples`.
- **Distributed** with data remaining on `multiple nodes` in a cluster.

Creating RDDs

Parallelizing a collection in driver program.

E.g., here is how to create a `parallelized` collection holding the numbers 1 to 5:

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

Here, `distData` is the new `RDD` created by calling `SparkContext's parallelize` method.

Creating RDDs

Referencing one dataset in an external storage system, like a `shared filesystem, HBase, HDFS, or any data source providing a Hadoop InputFormat`.

For example, text file RDDs can be created using the method `SparkContext's textFile`.

For the file (local path on the machine, hdfs://, s3n://, etc URI) the above method takes a URI and then reads it as a collection containing lines to produce the RDD.

```
distFile = sc.textFile("data.txt")
```

RDD Operations

RDDs support two types of operations: `transformations`, which create a new dataset from an existing one, and `actions`, which return a value to the driver program after running a computation on the dataset.

For example, `map` is a `transformation` that passes each dataset element through a function and returns a new RDD representing the results.

Similiarly, `reduce` is an `action` which aggregates all RDD elements by using some functions and then returns the final result to driver program.

More On RDD Operations

As a recap to RDD basics, consider the simple program shown below:

```
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
totalLength = lineLengths.reduce(lambda a, b: a + b)
```

The first line defines a `base RDD` from an `external file`.

The second line defines `lineLengths` as the result of a `map transformation`.

Finally, in the third line, we run `reduce`, which is an `action`.

Transformations

- Transformations are functions that use an RDD as the input and return one or more RDDs as the output.
- randomSplit, cogroup, join, reduceByKey, filter, and map are examples of few transformations.
- Transformations do not change the input RDD, but always create one or more new RDDs by utilizing the computations they represent.
- By using transformations, you incrementally create an RDD lineage with all the parent RDDs of the last RDD.
- Transformations are *lazy*, i.e. are not run immediately. Transformations are done on demand.
- Transformations are executed only after calling an action.

Examples Of Transformations

- **filter(func)**: Returns a new dataset (RDD) that are created by choosing the elements of the source on which the function returns true.

- **map(func)**: Passes each element of the RDD via the supplied function.
- **union()**: New RDD contains elements from source argument and RDD.
- **intersection()**: New RDD includes only common elements from source argument and RDD.
- **cartesian()**: New RDD cross product of all elements from source argument and RDD.

Actions

- Actions return concluding results of RDD computations.
- Actions trigger execution utilising lineage graph to load the data into original RDD, and then execute all intermediate transformations and write final results out to file system or return it to Driver program.
- `Count, collect, reduce, take, and first` are few actions in spark.

Example of Actions

- **count()**: Get the number of data elements in the RDD.
- **collect()**: Get all the data elements in an RDD as an array.
- **reduce(func)**: Aggregate the data elements in an RDD using this function which takes two arguments and returns one.
- **take (n)**: Fetch first n data elements in an RDD computed by driver program.
- **foreach(func)**: Execute function for each data element in RDD. usually used to update an accumulator or interacting with external systems.
- **first()**: Retrieves the first data element in RDD. It is similar to take(1).
- **saveAsTextFile(path)**: Writes the content of RDD to a text file or a set of text files to local file system/HDFS.

What is Dataframe ?

In general **DataFrames** can be defined as a data structure, which is tabular in nature. It represents rows, each of them consists of a number of observations.

Rows can have a variety of data formats `(heterogeneous)`, whereas a column can have data of the same data type `(homogeneous)`.

They mainly contain some `metadata` in addition to data like column and row names.

Why DataFrames ?

- DataFrames are widely used for processing a large collection of structured or semi-structured data
- They are having the ability to handle petabytes of data
- In addition, it supports a wide range of data format for reading as well as writing

As a conclusion DataFrame is data organized into named columns

Features of DataFrame

- Distributed
- Lazy Evals
- Immutable

Features Explained

- DataFrames are **Distributed** in Nature, which makes it fault tolerant and highly available data structure.
- **Lazy Evaluation** is an evaluation strategy which will hold the evaluation of an expression until its value is needed.
- DataFrames are **Immutable** in nature which means that it is an object whose state cannot be modified after it is created.

DataFrame Sources

For constructing a DataFrame a wide range of sources are available such as:

- `Structured data files`
- `Tables in Hive`
- `External Databases`
- `Existing RDDs`

Spark SQL

Spark introduces a programming module for structured data processing called **Spark SQL**.

It provides a programming abstraction called *DataFrame* and can act as distributed `SQL query engine`.

Features of Spark SQL

The main capabilities of using structured and semi-structured data, by **Spark SQL**. Such as:

- Provides DataFrame abstraction in Scala, Java, and Python.
- Spark SQL can read and write data from Hive Tables, JSON, and Parquet in various structured formats.
- Data can be queried by using Spark SQL.

For more details about Spark SQL refer the fresco course Spark SQL

Important classes of Spark SQL and DataFrames

- `pyspark.sql.SparkSession` :Main entry point for Dataframe SparkSQL functionality
- `pyspark.sql.DataFrame` :A distributed collection of data grouped into named columns
- `pyspark.sql.Column` : A column expression in a DataFrame.
- `pyspark.sql.Row` : A row of data in a DataFrame.
- `pyspark.sql.GroupedData` :Aggregation methods, returned by DataFrame.groupBy().

More On Classes

- `pyspark.sql.DataFrameNaFunctions` : Methods for handling missing data (null values).
- `pyspark.sql.DataFrameStatFunctions` : Methods for statistics functionality.
- `pyspark.sql.functions` : List of built-in functions available for DataFrame.
- `pyspark.sql.types` : List of data types available.
- `pyspark.sql.Window` : For working with window functions.

Creating a DataFrame demo

The entry point into all functionality in Spark is the **SparkSession class**.

To create a basic `SparkSession`, just use **SparkSession.builder**:

```
from pyspark.sql import SparkSession

spark = SparkSession \

    .builder \

    .appName("Data Frame Example") \

    .config("spark.some.config.option", "some-value") \

    .getOrCreate()
```

More On Creation

Import the `sql module` from *pyspark*

```


from pyspark.sql import *
```

```
Student = Row("firstName", "lastName", "age", "telephone")
s1 = Student('David', 'Julian', 22, 100000)
s2 = Student('Mark', 'Webb', 23, 658545)
StudentData=[s1,s2]
df=spark.createDataFrame(StudentData)
df.show()
```

```
from pyspark.sql import SparkSession
from pyspark import *
spark = SparkSession \
    .builder \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
passenger = Row("Name", "age", "source", "destination")
s1 = passenger('David', 22, 'London', 'Paris')
s2 = passenger('Steve', 22, 'New York', 'Sydney')
x = [s1,s2]
df1=spark.createDataFrame(x)
df1.show()
```

Result of show()

Once `show()` is executed we can view the following result in the `pyspark shell`

```
+---------+--------+---+---------+
|firstName|lastName|age|telephone|
+---------+--------+---+---------+
|    David|  Julian| 22|   100000|
|     Mark|    Webb| 23|   658545|
```

Data Sources

- Spark SQL supports operating on a variety of `data sources` through the DataFrame interface.
- A DataFrame can be operated on using `relational transformations` and can also be used to create a temporary view.
- Registering a DataFrame as a temporary view allows you to run `SQL queries` over its data.

- This chapter describes the general methods for `loading and saving data` using the Spark Data Sources.

Generic Load/Save Functions

In most of the cases, the default data source will be used for all operations.

```
df = spark.read.load("file path")


# Spark load the data source from the defined file path


df.select("column name", "column name").write.save("file name")


# The DataFrame is saved in the defined format


# By default it is saved in the Spark Warehouse
```

File path can be from *local machine* as well as from *HDFS*.

Manually Specifying Options

You can also manually specify the `data source` that will be used along with any `extra options` that you would like to pass to the data source.

Data sources fully qualified name is used to specify them, but for built-in sources, you can also use their short names `(json, parquet, jdbc, orc, libsvm, csv, text)`.

Specific File Formats

DataFrames which are loaded from any type of data can be converted to other types by using the syntax shown below.

A json file can be loaded:

```
df = spark.read.load("path of json file", format="json")
```

Apache Parquet

- Apache Parquet is a `columnar storage format` available to all projects in the `Hadoop ecosystem`, irrespective of the choice of the framework used for data processing, the model of data or programming language used.
- Spark SQL provides support for both `reading and writing Parquet files`.
- Automatic conversion to `nullable` occurs when one tries to write Parquet files, This is done due to `compatibility reasons`.

Reading A Parquet File

Here we are loading a `json` file into a dataframe.

```
df = spark.read.json("path of the file")
```

For saving the dataframe into `parquet` format.

```
df.write.parquet("parquet file name")
```

```
# Put your code here
from pyspark.sql import *
spark = SparkSession.builder.getOrCreate()
df = spark.read.json("emp.json")
df.show()
df.write.parquet("Employees")
df.createOrReplaceTempView("data")
res = spark.sql("select age,name,stream from data where stream='JAVA'")
res.show()
res.write.parquet("JavaEmployees")
```

Verifying The Result

We can verify the result by loading in `Parquet` format.

```
pf = spark.read.parquet("parquet file name")
```

Here we are reading in `Parquet` format.

To view the the `DataFrame` use `show()` method.

Why Parquet File Format ?

- Parquet stores nested data structures in a `flat columnar format`.
- On comparing with the traditional way instead of storing in `row-oriented way` in parquet is more `efficient`
- Parquet is the choice of `Big data` because it serves both needs, `efficient` and `performance` in both `storage` and `processing`.

Why Parquet File Format ?

- Parquet stores nested data structures in a `flat columnar format`.
- On comparing with the traditional way instead of storing in `row-oriented way` in parquet is more `efficient`

- Parquet is the choice of `Big data` because it serves both needs, `efficient` and `performance` in both `storage` and `processing`.

Advanced Concepts in Data Frame

In this chapter, you will learn how to perform some advanced operations on `DataFrames`.

Throughout the chapter, we will be focusing on `csv` files.

Reading Data From A CSV File

What is a CSV file?

- **CSV** is a file format which allows the user to store the data in tabular format.
- CSV stands for `comma-separated values`.
- It's data fields are most often `separated`, or `delimited`, by a `comma`.

CSV Loading

To load a **csv** data set user has to make use of `spark.read.csv` method to load it into a DataFrame.

Here we are loading a **football player dataset** using the spark `csvreader`.

```
df = spark.read.csv("path-of-file/fifa_players.csv", inferSchema = True, header = True)
```

CSV Loading

inferSchema (default false): From the data, it infers the input schema automatically.

header (default false): Using this it inherits the first line as column names.

To verify we can run `df.show(2)`.

The argument `2` will display the first two rows of the resulting DataFrame.

For every example from now onwards we will be using football player DataFrame

Schema of DataFrame

What is meant by **schema**?

It's just the `structure` of the DataFrame.

To check the schema one can make use of `printSchema` method.

It results in different `columns` in our DataFrame, along with the `datatype` and the `nullable` conditions.

How To Check The Schema

To check the `schema` of the loaded `csv` data.

```
df.printSchema()
```

Once executed we will get the following result.

```
root
 |-- ID: integer (nullable = true)
 |-- Name: string (nullable = true)
 |-- Age: integer (nullable = true)
 |-- Nationality: string (nullable = true
 |-- Overall: integer (nullable = true)
 |-- Potential: integer (nullable = true)
 |-- Club: string (nullable = true)
 |-- Value: string (nullable = true)
 |-- Wage: string (nullable = true)
 |-- Special: integer (nullable = true)
```

Column Names and Count (Rows and Column)

For finding the column names, count of the number of rows and columns we can use the following methods.

For Column names

```
df.columns
['ID', 'Name', 'Age', 'Nationality', 'Overall', 'Potential', 'Club', 'Value',
'Wage', 'Special']
```

Row count

```
df.count()
17981
```

Column count

```
len(df.columns)
```

```
10
```

Describing a Particular Column

To get the `summary` of any particular column make use of `describe` method.

This method gives us the `statistical summary` of the given `column`, if not specified, it provides the `statistical summary` of the `DataFrame`.

```
df.describe('Name').show()
```

The result we will be as shown below.

```
+-------+-------------+
|summary|         Name|
+-------+-------------+
|  count|        17981|
|   mean|         null|
| stddev|         null|
|    min|     A. Abbas|
|    max|Óscar Whalley|
+-------+-------------+
```

Advanced Concepts in Data Frame

In this chapter, you will learn how to perform some advanced operations on `DataFrames`.

Throughout the chapter, we will be focusing on `csv` files.

Reading Data From A CSV File

What is a CSV file?

- **CSV** is a file format which allows the user to store the data in tabular format.
- CSV stands for `comma-separated values`.
- It's data fields are most often `separated`, or `delimited`, by a `comma`.

CSV Loading

To load a **csv** data set user has to make use of `spark.read.csv` method to load it into a DataFrame.

Here we are loading a **football player dataset** using the spark `csvreader`.

```
df = spark.read.csv("path-of-file/fifa_players.csv", inferSchema = True, header = True)
```

CSV Loading

inferSchema (default false): From the data, it infers the input schema automatically.

header (default false): Using this it inherits the first line as column names.

To verify we can run `df.show(2)`.

The argument `2` will display the first two rows of the resulting DataFrame.

For every example from now onwards we will be using football player DataFrame

Schema of DataFrame

What is meant by **schema**?

It's just the `structure` of the DataFrame.

To check the schema one can make use of `printSchema` method.

It results in different `columns` in our DataFrame, along with the `datatype` and the `nullable` conditions.

How To Check The Schema

To check the `schema` of the loaded `csv` data.

```
df.printSchema()
```

Once executed we will get the following result.

```
root
 |-- ID: integer (nullable = true)
 |-- Name: string (nullable = true)
 |-- Age: integer (nullable = true)
 |-- Nationality: string (nullable = true
 |-- Overall: integer (nullable = true)
 |-- Potential: integer (nullable = true)
 |-- Club: string (nullable = true)
 |-- Value: string (nullable = true)
 |-- Wage: string (nullable = true)
```

```
 |-- Special: integer (nullable = true)
```

Column Names and Count (Rows and Column)

For finding the column names, count of the number of rows and columns we can use the following methods.

For Column names

```
df.columns
```

```
['ID', 'Name', 'Age', 'Nationality', 'Overall', 'Potential', 'Club', 'Value', 'Wage', 'Special']
```

Row count

```
df.count()
```

```
17981
```

Column count

```
len(df.columns)
```

```
10
```

Describing a Particular Column

To get the `summary` of any particular column make use of `describe` method.

This method gives us the `statistical summary` of the given `column`, if not specified, it provides the `statistical summary` of the `DataFrame`.

```
df.describe('Name').show()
```

The result we will be as shown below.

```
+-------+-------------+
|summary|         Name|
+-------+-------------+
|  count|        17981|
|   mean|         null|
| stddev|         null|
|    min|     A. Abbas|
|    max|Óscar Whalley|
```

```
+-------+------------+
```

Describing A Different Column

Now try it on some other column.

```
df.describe('Age').show()
```

Observe the various `Statistical Parameters`.

```
+-------+------------------+
|summary|               Age|
+-------+------------------+
|  count|             17981|
|   mean|25.144541460430453|
| stddev| 4.614272345005111|
|    min|                16|
|    max|                47|
+-------+------------------+
```

Selecting Multiple Columns

For selecting particular columns from the DataFrame, one can use the select method.

Syntax for performing selection operation is:

```
df.select('Column name 1,'Column name 2',.....,'Column name n').show()
```

**show() is optional **

One can load the result into another DataFrame by simply equating.

ie

```
dfnew=df.select('Column name 1,'Column name 2',.....,'Column name n')
```

Selection Operation

Selecting the column `ID` and `Name` and loading the result to a new `DataFrame`.

```
dfnew=df.select('ID','Name')
```

Verifying the result

```
dfnew.show(5)
+------+----------------+
|    ID|            Name|
+------+----------------+
| 20801|Cristiano Ronaldo|
|158023|         L. Messi|
|190871|           Neymar|
|176580|        L. Suárez|
|167495|        M. Neuer|
+------+----------------+
```

only showing top 5 rows

Filtering Data

For filtering the data `filter` command is used.

```
df.filter(df.Club=='FC Barcelona').show(3)
```

The result will be as follows:

```
+------+----------+---+-----------+-------+---------+------------+------+-----
+-------+
|    ID|      Name|Age|Nationality|Overall|Potential|        Club| Value| Wage|
Special|
+------+----------+---+-----------+-------+---------+------------+------+-----
+-------+
|158023|  L. Messi| 30|  Argentina|     93|       93|FC Barcelona| €105M|€565K|
2154|
|176580| L. Suárez| 30|    Uruguay|     92|       92|FC Barcelona|  €97M|€510K|
2291|
|168651|I. Rakitić| 29|    Croatia|     87|       87|FC Barcelona|€48.5M|€275K|
2129|
+------+----------+---+-----------+-------+---------+------------+------+-----
+-------+
```

only showing top 3 rows since we had given 3 in the show() as the argument

Verify the same by your own

To Filter our data based on multiple conditions (AND or OR)

```
df.filter((df.Club=='FC Barcelona') & (df.Nationality=='Spain')).show(3)

+------+--------------+---+-----------+-------+---------+------------+------
+-----+-------+

|    ID|          Name|Age|Nationality|Overall|Potential|        Club| Value|
Wage|Special|

+------+--------------+---+-----------+-------+---------+------------+------
+-----+-------+

|152729|         Piqué| 30|      Spain|     87|       87|FC Barcelona|€37.5M|
€240K|   1974|

|    41|        Iniesta| 33|      Spain|     87|       87|FC Barcelona|€29.5M|
€260K|   2073|

|189511|Sergio Busquets| 28|      Spain|     86|       86|FC Barcelona|  €36M|
€250K|   1998|

+------+--------------+---+-----------+-------+---------+------------+------
+-----+-------+
```

only showing top 3 rows

In a similar way we can use other `logical operators`.

Sorting Data (OrderBy)

To sort the data use the `OrderBy` method.

In pyspark in default, it will sort in `ascending` order but we can change it into `descending` order as well.

```
df.filter((df.Club=='FC Barcelona') &
(df.Nationality=='Spain')).orderBy('ID',).show(5)
```

To sort in descending order:

```
df.filter((df.Club=='FC Barcelona') &
(df.Nationality=='Spain')).orderBy('ID',ascending=False).show(5)
```

Sorting

The result of the first order by operation results in the following output.

```
+------+--------------+---+-----------+-------+---------+------------+------
+-----+-------+
```

```
|    ID|           Name|Age|Nationality|Overall|Potential|        Club| Value|
Wage|Special|
+------+---------------+---+-----------+-------+---------+------------+------
+-----+-------+
|    41|        Iniesta| 33|      Spain|     87|       87|FC Barcelona|€29.5M|
€260K|   2073|
|152729|          Piqué| 30|      Spain|     87|       87|FC Barcelona|€37.5M|
€240K|   1974|
|189332|     Jordi Alba| 28|      Spain|     85|       85|FC Barcelona|€30.5M|
€215K|   2206|
|189511|Sergio Busquets| 28|      Spain|     86|       86|FC Barcelona|  €36M|
€250K|   1998|
|199564|  Sergi Roberto| 25|      Spain|     81|       86|FC Barcelona|€19.5M|
€140K|   2071|
+------+---------------+---+-----------+-------+---------+------------+------
+-----+-------+
```

only showing top 5 rows

Random Data Generation

Random Data generation is useful when we want to test algorithms and to implement new ones.

In Spark under `sql.functions` we have methods to generate random data. e.g., uniform `(rand)`, and standard normal `(randn)`.

```
from pyspark.sql.functions import rand, randn
df = sqlContext.range(0, 7)
df.show()
```

The output will be as follows

```
+---+
| id|
+---+
|  0|
|  1|
|  2|
|  3|
+---+
```

More on Random Data

By using uniform distribution and normal distribution generate two more columns.

```
df.select("id", rand(seed=10).alias("uniform"),
randn(seed=27).alias("normal")).show()
```

```
+---+------------------+------------------+
| id|           uniform|            normal|
+---+------------------+------------------+
|  0|0.41371264720975787| 0.5888539012978773|
|  1| 0.1982919638208397|0.06157382353970104|
|  2|0.12030715258495939| 1.0854146699817222|
|  3|0.44292918521277047|-0.4798519469521663|
+---+------------------+------------------+
```

Summary and Descriptive Statistics

The `first operation` to perform after importing data is to get some `sense` of what it looks like.

The function `describe` returns a DataFrame containing information such as number of non-null entries (count), `mean`, `standard deviation`, and `minimum` and `maximum` value for each numerical column.

Summary and Descriptive Statistics

```
df.describe('uniform', 'normal').show()
```

The result is as follows:

```
+-------+------------------+--------------------+
|summary|           uniform|              normal|
+-------+------------------+--------------------+
|  count|                10|                  10|
|   mean| 0.3841685645682706|-0.15825812884638607|
| stddev|0.31309395532409323|   0.963345903544872|
|    min|0.03650707717266999| -2.1591956435415334|
|    max| 0.8898784253886249|  1.0854146699817222|
+-------+------------------+--------------------+
```

Descriptive Statistics

For a quick review of a column `describe` works fine.

In the same way, we can also make use of some standard `statistical functions` also.

```
from pyspark.sql.functions import mean, min, max

df.select([mean('uniform'), min('uniform'), max('uniform')]).show()

+------------------+-------------------+------------------+
|      avg(uniform)|       min(uniform)|      max(uniform)|
+------------------+-------------------+------------------+
|0.3841685645682706|0.03650707717266999|0.8898784253886249|
+------------------+-------------------+------------------+
```

Sample Co-Variance and Correlation

In statistics **Co-Variance** comeans how one `random variable` changes with respect to other.

`Positive` value indicates a trend in increase when the other increases.

`Negative` value indicates a trend in decrease when the other increases.

The sample co-variance of two columns of a DataFrame can be calculated as follows:

More On Co-Variance

```
from pyspark.sql.functions import rand

df = sqlContext.range(0, 10).withColumn('rand1',
rand(seed=10)).withColumn('rand2', rand(seed=27))

df.stat.cov('rand1', 'rand2')

0.031109767020625314
```

From the above we can infer that `co-variance` of two random columns is near to `zero`.

Correlation

Correlation provides the statistical dependence of two random variables.

```
df.stat.corr('rand1', 'rand2')

0.30842745432650953
```

Two randomly generated columns have `low correlation value`.

Cross Tabulation (Contingency Table)

Cross Tabulation provides a `frequency distribution table` for a given set of variables.

One of the powerful tool in statistics to observe the statistical `independence` of variables.

Consider an example

```
# Create a DataFrame with two columns (name, item)

names = ["Alice", "Bob", "Mike"]

items = ["milk", "bread", "butter", "apples", "oranges"]

df = sqlContext.createDataFrame([(names[i % 3], items[i % 5]) for i in
range(100)], ["name", "item"])
```

Contingency Table)

For applying the `cross tabulation` we can make use of the crosstab method.

```
df.stat.crosstab("name", "item").show()

+---------+------+-----+------+----+-------+

|name_item|apples|bread|butter|milk|oranges|

+---------+------+-----+------+----+-------+

|      Bob|     6|    7|     7|   6|      7|

|     Mike|     7|    6|     7|   7|      6|

|    Alice|     7|    7|     6|   7|      7|

+---------+------+-----+------+----+-------+
```

`Cardinality` of columns we run `crosstab` on cannot be too big.

```
# Put your code here
from pyspark.sql import *
from pyspark import SparkContext
from pyspark.sql.functions import rand, randn
from pyspark.sql import SQLContext
from pyspark.sql.types import FloatType
spark = SparkSession.builder.getOrCreate()
df1 = Row("1","2")
sqlContext = SQLContext(spark)
df = sqlContext.range(0, 10).withColumn('rand1', rand(seed=10)).withColumn('ra
nd2', rand(seed=27))
a = df.stat.cov('rand1', 'rand2')
b = df.stat.corr('rand1', 'rand2')

s1 = df1("Co-variance",a)
s2 = df1("Correlation",b)
x=[s1,s2]
```

```
df2 = spark.createDataFrame(x)
df2.show()
df2.write.parquet("Result")
```

What is Spark SQL ?

Spark SQL brings native support for `SQL` to Spark.

Spark SQL blurs the lines between `RDD's` and `relational tables`.

By integrating these powerful features Spark makes it easy for developers to use SQL commands for querying external data with complex analytics, all within in a single application.

Performing SQL Queries

We can also pass `SQL queries` directly to any DataFrame.

For that, we need to create a table from the DataFrame using the `registerTempTable` method.

After that use `sqlContext.sql()` to pass the SQL queries.

Apache Hive

The `Apache Hive` data warehouse software allows `reading`, `writing`, and managing `large datasets` residing in distributed storage and queried using `SQL syntax`.

Features of Apache Hive

Apache Hive is built on *top* of Apache Hadoop.

The below mentioned are the features of Apache Hive.

- Apache Hive is having tools to allow easy and quick access to data using SQL, thus enables data warehousing tasks such like extract/transform/load `(ETL)`, reporting, and data analysis.
- Mechanisms for imposing structure on a variety of `data formats`.
- Access to files stored either directly in `Apache HDFS` or in other data storage systems such as `Apache HBase`.

Features Of Apache Hive

- Query execution via `Apache Tez`,`Apache Spark`, or `MapReduce`.
- A procedural language with `HPL-SQL`.
- Sub-second query retrieval via `Hive LLAP`, `Apache YARN` and `Apache Slider`.

What Hive Provides ?

Apache Hive provides the standard SQL functionalities, which includes many of the later `SQL:2003` and `SQL:2011` features for analytics.

We can extend Hive's SQL with the user code by using user-defined functions `(UDFs)`, user-defined aggregates `(UDAFs)`, and user-defined table functions `(UDTFs)`.

Hive comes with built-in connectors for `comma and tab-separated values (CSV/TSV) text files`, `Apache Parquet`,`Apache ORC`, and other formats.

Spark SQL supports `reading` and `writing` data stored in `Hive`. # Connecting Hive From Spark

When working with Hive, one must instantiate `SparkSession` with `Hive support`, including connectivity to a persistent `Hive metastore`, support for `Hive serdes`, and `Hive user-defined functions`.

How To Enable Hive Support

```
from os.path import expanduser, join, abspath


from pyspark.sql import SparkSession

from pyspark.sql import Row
```

```
#warehouse_location points to the default location for managed databases and
tables


warehouse_location = abspath('spark-warehouse')


spark = SparkSession \
    .builder \
    .appName("Python Spark SQL Hive integration example") \
    .config("spark.sql.warehouse.dir", warehouse_location) \
    .enableHiveSupport() \
    .getOrCreate()
```

reating Hive Table From Spark

We can easily create a table in `hive warehouse` programmatically from Spark.

The syntax for creating a table is as follows:

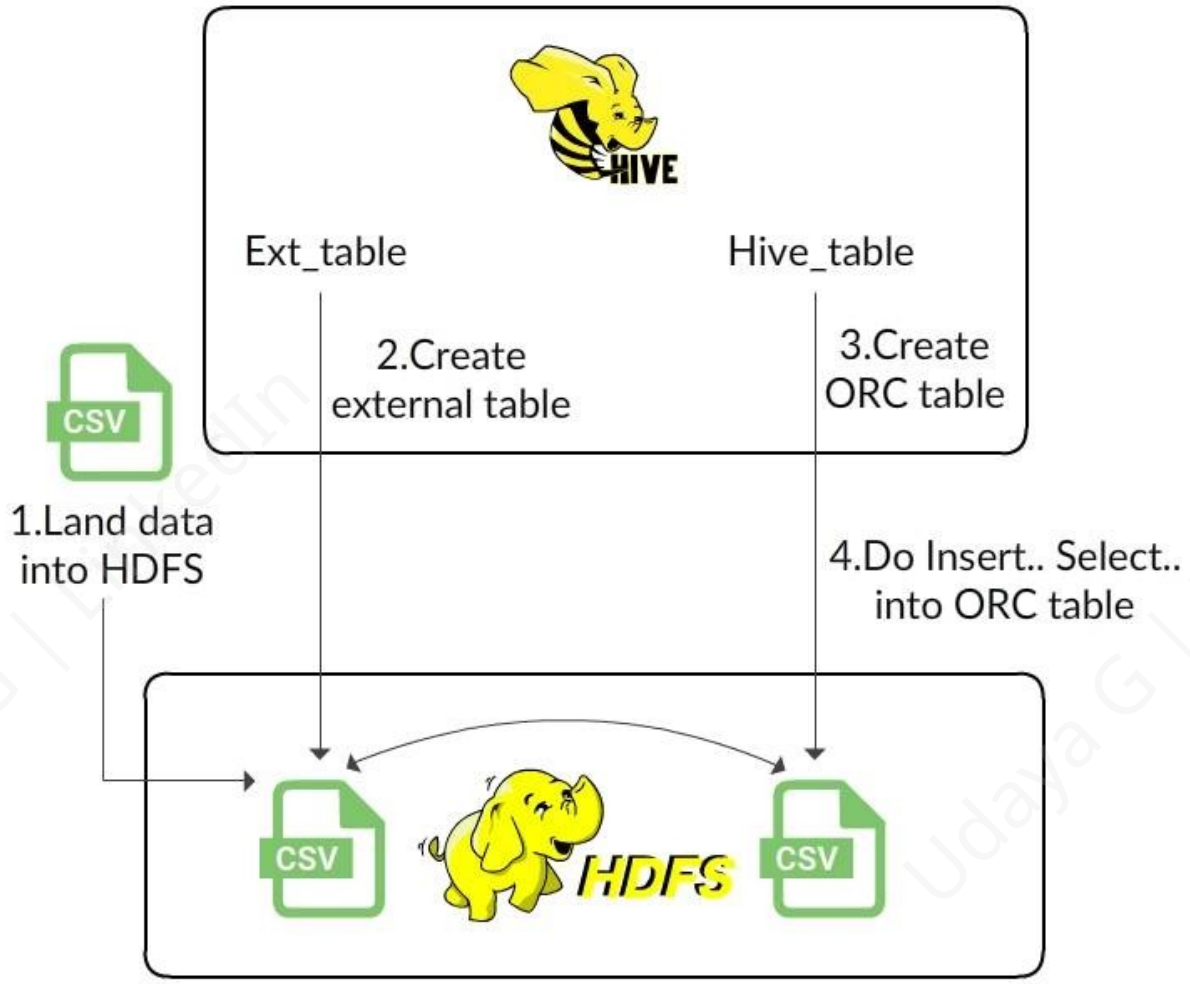
spark.sql("CREATE TABLE IF NOT EXISTS table_name(column_name_1 DataType,column_name_2 DataType,.....,column_name_n DataType) USING hive")

To load a DataFrame into a table.

```
df.write.insertInto("table name",overwrite = True)
```

Now verify the result by using the select statement.

Hive External Table



- External tables are used to store data `outside` the hive.
- Data needs to remain in the `underlying location` even after the user drop the table.

Handling External Hive Tables From Apache Spark

First, create an `external table` in the hive by specifying the location.

One can create an external table in the hive by running the following query on `hive shell`.

hive> create external table table_name(column_name1 DataType,column_name2 DataType,.....,column_name_n DataType) STORED AS Parquet location ' path of external table';

- The table is created in `Parquet schema`.
- The table is saved in the `hdfs` directory.

Loading Data From Spark To The Hive Table

We can load data to `hive table` from the `DataFrame`.

For doing the same `schema` of both `hive table` and the `DataFrame` should be equal.

Let us take a sample CSV file.

We can read the `csv` the file by making use of `spark csv reader`.

```
df = spark.read.csv("path-of-file", inferSchema = True, header = True)
```

The `schema` of the DataFrame will be same as the schema of the CSV file itself.

Data Loading To External Table

For loading the data we have to save the dataframe in external hive table location.

```
df.write.mode('overwrite').format("format").save("location")
```

Since our hive external table is in `parquet format` in place of format we have to mention 'parquet'.

The location should be `same` as the hive external table location in hdfs directory.

If the `schema` is matching then data will load `automatically` to the hive table.

By querying the hive table we can verify it.

What is HBase ?

HBase is a `distributed` column-oriented data store built on top of `HDFS`.

HBase is an Apache open source project whose goal is to provide `storage` for the Hadoop Distributed Computing.

Data is logically organized into `tables`, `rows` and `columns`.

More On HBase

- HBase features compression, in-memory operation, and Bloom filters on a per-column basis as outlined in the original `Bigtable` paper.
- Tables in HBase can serve as the `input` and `output` for Map Reduce jobs run in Hadoop, and may be accessed through the `Java API` but also through `REST`, `Avro` or `Thrift gateway APIs`.
- It is a column-oriented key-value data store and has been idolized widely because of its `lineage` with `Hadoop` and `HDFS`.
- HBase runs on `top` of HDFS and is well-suited for `faster` read and write operations on large datasets with high `throughput` and low `input/output latency`.

How To Connect Spark and HBase

- To connect we require `hdfs`,`Spark` and `HBase` installed in the local machine.
- Make sure that your `versions` are `matching` with each other.
- Copy all the `HBase jar` files to the `Spark lib` folder.
- Once done set the `SPARK_CLASSPATH` in `spark-env.sh` with `lib`.

Building A Real-Time Data Pipeline

Real Time Pipeline using HDFS,Spark and HBase

Various Stages

It has `4 main stages` which includes:

- Transformation
- Cleaning
- Validation
- Writing of the data received from the various sources

Transformation And Cleaning

Data Transformation

- This is an entry point for the `streaming application.`
- Here the operations related to `normalization` of data are performed.
- Transformation of data can be performed by using built-in functions like `map`, `filter`, `foreachRDD` etc.

Data Cleaning

- During `preprocessing` cleaning is very important.
- In this stage, we can use `custom built libraries` for cleaning the data.

Validation And Writing

Data Validation

In this stage, we can `validate` the data with respect to some `standard validations` such as length, patterns and so on.

Writing

At last, data passed from the `previous three stages` is passed on to the `writing application` which simply writes this final set of data to `HBase` for further data analysis.

Spark In Real World

Uber – the online taxi company is an apt example for Spark. They are gathering terabytes of event data from its various users.

- Uses `Kafka, Spark Streaming, and HDFS`, to build a continuous `ETL pipeline`.
- Convert raw unstructured data into structured data as it is collected.
- Uses it further complex analytics and optimization of operations.

Spark In Real World

Pinterest – Uses a Spark ETL pipeline

- Leverages Spark Streaming to gain `immediate insight` into how users all over the world are `engaging` with `Pins` in real time.
- Can make more relevant recommendations as people navigate the site.
- Recommends related Pins.
- Determine which products to `buy`, or `destinations` to visit.

Spark In Real World

Conviva – 4 million video feeds per month.

- Conviva is using Spark for reducing the customer churn by `managing live video traffic` and `optimizing video streams`.
- They maintain a consistently smooth high-quality viewing experience.

Spark In Real World

Capital One – makes use of Spark and data science algorithms for a better understanding of its customers.

- Developing the next generation of `financial products and services`.
- Find attributes and patterns of increased `probability for fraud`.

Netflix – Movie recommendation engine from user data.

- User data is also used for `content creation`

```
# Put your code here
from pyspark.sql import *
spark = SparkSession.builder.getOrCreate()
df = Row("ID","Name","Age","AreaofInterest")
s1 = df("1","Jack",22,"Data Science")
s2 = df("2","Leo",21,"Data Analytics")
s3 = df("3","Luke",24,"Micro Services")
s4 = df("4","Mark",21,"Data Analytics")
x = [s1,s2,s3,s4]
df1 = spark.createDataFrame(x)
df3 = df1.describe("Age")
df3.show()
df3.write.parquet("Age")
df1.createOrReplaceTempView("data")
df4 = spark.sql("select ID,Name,Age from data order by ID desc")
df4.show()
df4.write.parquet("NameSorted")
```