SPARK

It is a distributed computing engine which distributes our data to process it.

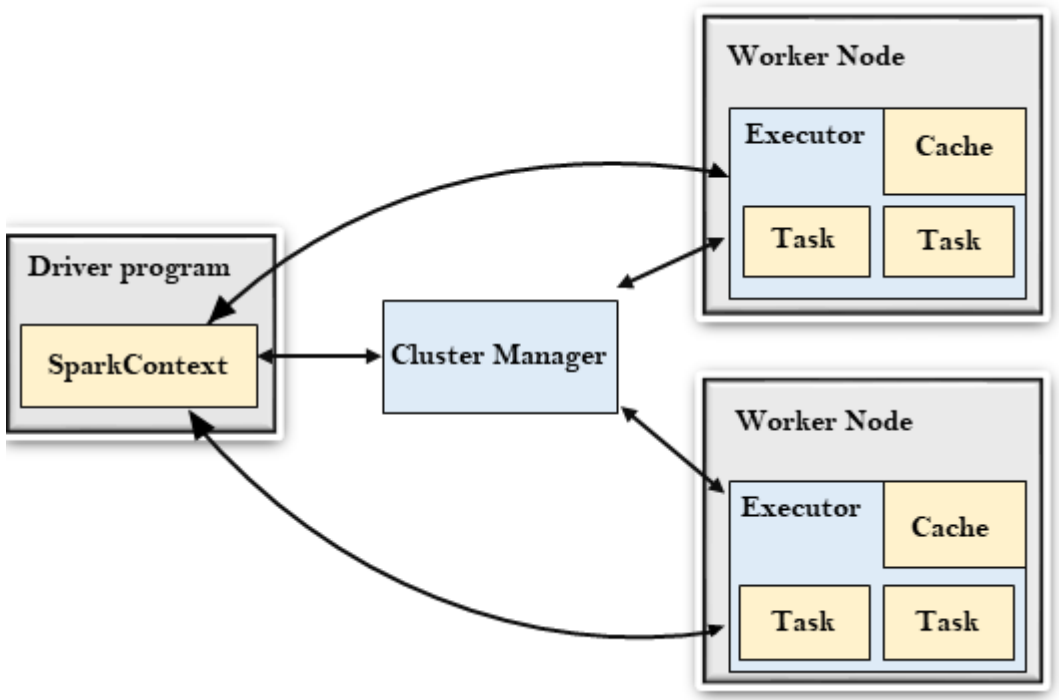
For eg:

If we have 1 GB of data then to process it we have 2 options-

- Using only one machine in which we can increase the resources to work efficiently. In this, we can increase the RAM, ROM etc upto a particular limit.
- Using multiple machines and connect together and form a cluster. There is no limit for RAM, ROM, etc.

So, we prefer the 2nd option.

SPARK ARCHITECTURE (Master-Slave Architecture)



Working

First, the cluster manager creates a driver node (computer) on our cluster. Driver node is ready.

Let us assume if I submit a code then it will go to the driver node. It will break that code in the form of jobs/task and will hand over all the information to cluster manager.

Then, the driver program requests cluster manager to create 2 worker nodes and all the breakdown will go to these workers. These 2 workers will actually execute those transformations.

BENEFITS(Why replace Hadoop?)

- In-Memory Computation: Hadoop was taking disk memory and was taking a long time to come back so we used spark which processes all the data in memory.
- Lazy Evaluation: Suppose there are transformations in the DataFrame then these transformations are stored in a logical plan and it will optimize the plan, create a plan and then once we hit the action it will execute the logical plan.
  Action examples: spark.show(), spark.display()
- Fault Tolerant: We get the ability to trace back all the transformations.
- Partitioning: Partitions and distributes data to the cluster of the machine.

Each job is divided into stages and each stages are divided into various tasks.

APIs are available in Spark.

API- It allows us to write our code in native language such as Python, Scala, SQL, R

Python + Spark -> PySpark

TABS

Workspace- we store all our notebooks

Catalog- we will be saving or uploading the data within this

Workflows- we build connections between notebooks

Compute- we create our cluster so that transformations can be carried out easily

INSIDE NOTEBOOK

Connect- used to create cluster and attach to the notebook

Green light- ready to work with this cluster

Markdown- to create headings, subheadings, add comments

- %md- it is a magic command
- ###- to create heading
  Equivalent to h3 command in HTML
  Click Shift+Enter to quit
- ####- subheading

DATA READING

For reading csv file:

```
df=spark.read.format('csv').option('inferSchema',True).option('header',True).
load('/FileStore/tables/BigMart_Sales.csv')
```

- spark.read: dataframe reader API within PySpark
- inferSchema: automatically infers schema by looking at some records and assumed the best fit regarding data types
- spark will go and read the data and then make assumptions of data types

`dbutils.fs.ls('/FileStore/tables/')`: gives information about the path

df.show(): shows us data available in that particular DataFrame

`df.display()`: shows output in more neat form

For reading json file:

```
df_json = spark.read.format('json').option('inferSchema',True)\
                    .option('header',True)\
                    .option('multiLine',False)\
                    .load('/FileStore/tables/drivers.json')


df_json.display()
```

Schema-DDL and StructType()

`df.printSchema()`: to print the schema

DDL

```
my_ddl_schema = '''
```

```
                    Item_Identifier STRING,
                    Item_Weight STRING,
                    Item_Fat_Content STRING,
                    Item_Visibility DOUBLE,
                    Item_Type STRING,
                    Item_MRP DOUBLE,
                    Outlet_Identifier STRING,
                    Outlet_Establishment_Year INT,
                    Outlet_Size STRING,
                    Outlet_Location_Type STRING,
                    Outlet_Type STRING,
                    Item_Outlet_Sales DOUBLE

                '''
```

: to change the datatype

```
df = spark.read.format('csv')\
            .schema(my_ddl_schema)\
            .option('header',True)\
            .load('/FileStore/tables/BigMart_Sales.csv')
df.display()
df.printSchema()
```

OR

StructType() Schema

```
from pyspark.sql.types import *
from pyspark.sql.functions import *
```

mystrct_schema=StructType([StructField('Item_Identifier',StringType(),True)])

df=spark.read.format('csv').schema(mystrct_schema).option('header',True).load('/FileStore/…)

TRANSFORMATIONS

Select (selectively pull the commands)

```
df.select(col('Item_Identifier'),col('Item_Weight'),col('Item_Fat_Content')).
display()
```

Alias

```
df.select(col('Item_Identifier').alias('Item_ID')).display()
```

Filter

Scenario - 1

```
df.filter(col('Item_Fat_Content')=='Regular').display()
```

Scenario - 2

```
df.filter((col('Item_Type') == 'Soft Drinks') &
(col('Item_Weight')<10)).display()
```

Scenario - 3

```
df.filter((col('Outlet_Size').isNull()) &
(col('Outlet_Location_Type').isin('Tier 1','Tier 2'))).display()
```

withColumnRenamed (used for renaming)

```
df.withColumnRenamed('Item_Weight','Item_Wt').display()
```

withColumn (add new column or modify a column)

Scenario - 1

```
df = df.withColumn('flag',lit("new"))

df.display()

df.withColumn('multiply',col('Item_Weight')*col('Item_MRP')).display()
```

Scenario - 2

```
df=df.withColumn('Item_Fat_Content',regexp_replace(col('Item_Fat_Content'),"R
egular","Reg"))\
.withColumn('Item_Fat_Content',regexp_replace(col('Item_Fat_Content'),"Low
Fat","Lf"))

df.display()
```

{ used for modification}

- regxp_replace(): function to replace column values

Type Casting

```
df = df.withColumn('Item_Weight', col('Item_Weight').cast(StringType()))
```

```
df.printSchema()
```

Sort/ OrderBy()

Scenario - 1

```
df.sort(col('Item_Weight').desc()).display()
```

Scenario - 2

```
df.sort(col('Item_Visibility').asc()).display()
```

Scenario - 3

```
df.sort(['Item_Weight','Item_Visibility'],ascending = [0,0]).display()
```

Scenario - 4

```
df.sort(['Item_weight','Item_Visibility'], ascending = [0,1]).display()
```

Limit

```
df.limit(10).display()
```

Drop

Scenario-1

```
df.drop('Item_Visibility').display()
```

Scenario-2

```
df.drop('Item_Visibility','Item_Type').display()
```

Drop Duplicates

`df.dropDuplicates().display()` - all duplicates are removed

`df.drop_duplicates(subset=['Item_Type']).display()` – to drop a particular column

```
df.distinct().display()
```

Union AND UnionByName

Preparing DataFrame

```
data1 = [('1','kad'),

        ('2','sid')]

schema1 = 'id STRING, name STRING'

df1 = spark.createDataFrame(data1,schema1)

data2 = [('3','rahul'),

        ('4','jas')]

schema2 = 'id STRING, name STRING'

df2 = spark.createDataFrame(data2,schema2)

df1.display()

df2.display()
```

Union (it displays as it is defined in the schema)

```
df1.union(df2).display()
```

UnionByName(it takes care of all the mappings)

```
df1.unionByName(df2).display()
```

String Functions

initcap(): first alphabet of every word becomes capital

- df.select(initcap('Item_Type'))

lower(): converts the words into lower case

- df.select(lower('Item_Type'))

upper(): converts the words into upper case

- df.select(upper('Item_Type'))

Date Functions

Current_Date

```
df = df.withColumn('curr_date',current_date())

df.display()
```

Date_Add()

```
df = df.withColumn('week_after',date_add('curr_date',7))

df.display()
```

Date_Sub()

```
df.withColumn('week_before',date_sub('curr_date',7)).display()

                          OR

df = df.withColumn('week_before',date_add('curr_date',-7))

df.display()
```

DateDiff

```
df = df.withColumn('datediff',datediff('week_after','curr_date'))

df.display()
```

Date_Format()

```
df = df.withColumn('week_before',date_format('week_before','dd-MM-yyyy'))

df.display()
```

HANDLING NULLS

Dropping Nulls

```
df.dropna('all').display()

df.dropna('any').display()
```

- all: will drop all the records which have null in all the columns
- any: will drop all the records having null in any column

```
df.dropna(subset=['Outlet_Size']).display()
```

Filling Nulls

```
df.fillna('NotAvailable').display()
```

```
df.fillna('NotAvailable',subset=['Outlet_Size']).display()
```

Split AND Indexing

Split

```
df.withColumn('Outlet_Type',split('Outlet_Type',' ')).display()
```

Indexing

```
df.withColumn('Outlet_Type',split('Outlet_Type',' ')[1]).display()
```

Explode

```
df_exp = df.withColumn('Outlet_Type',split('Outlet_Type',' '))
```

```
df_exp.display()
```

```
df_exp.withColumn('Outlet_Type',explode('Outlet_Type')).display()
```

```
df_exp.display()
```

Array_Contains

```
df_exp.withColumn('Type1_flag',array_contains('Outlet_Type','Type1')).display
()
```

GroupBy

Scenario - 1

```
df.groupBy('Item_Type').agg(sum('Item_MRP')).display()
```

Scenario - 2

```
df.groupBy('Item_Type').agg(avg('Item_MRP')).display()
```

Scenario - 3

```
df.groupBy('Item_Type','Outlet_Size').agg(sum('Item_MRP').alias('Total_MRP'))
.display()
```

Scenario - 4

```
df.groupBy('Item_Type','Outlet_Size').agg(sum('Item_MRP'),avg('Item_MRP')).di
splay()
```

Collect_List

```
data = [('user1','book1'),

        ('user1','book2'),

        ('user2','book2'),

        ('user2','book4'),

        ('user3','book1')]

schema = 'user string, book string'

df_book = spark.createDataFrame(data,schema)

df_book.display()

df_book.groupBy('user').agg(collect_list('book')).display()

df.select('Item_Type','Outlet_Size','Item_MRP').display()
```

Pivot

```
df.groupBy('Item_Type').pivot('Outlet_Size').agg(avg('Item_MRP')).display()
```

When_Otherwise

```
df=df.withColumn('veg_flag',when(col('Item_Type')=='Meat','Non-Veg').otherwis
e('Veg'))


df.withColumn('veg_exp_flag',when(((col('veg_flag')=='Veg') &
(col('Item_MRP')<100)),'Veg_Inexpensive')\

                            .when((col('veg_flag')=='Veg') &
(col('Item_MRP')>100),'Veg_Expensive')\

                            .otherwise('Non_Veg')).display()
```

Joins

```
dataj1 = [('1','gaur','d01'),

          ('2','kit','d02'),

          ('3','sam','d03'),

          ('4','tim','d03'),

          ('5','aman','d05'),

          ('6','nad','d06')]

schemaj1 = 'emp_id STRING, emp_name STRING, dept_id STRING'
```

```
df1 = spark.createDataFrame(dataj1,schemaj1)

dataj2 = [('d01','HR'),
          ('d02','Marketing'),
          ('d03','Accounts'),
          ('d04','IT'),
          ('d05','Finance')]

schemaj2 = 'dept_id STRING, department STRING'

df2 = spark.createDataFrame(dataj2,schemaj2)

df1.display()

df2.display()
```

Inner Join

```
df1.join(df2, df1['dept_id']==df2['dept_id'],'inner').display()
```

Left Join

```
df1.join(df2,df1['dept_id']==df2['dept_id'],'left').display()
```

Right Join

```
df1.join(df2,df1['dept_id']==df2['dept_id'],'right').display()
```

Anti Join

```
df1.join(df2,df1['dept_id']==df2['dept_id'],'anti').display()
```

Window Functions

```
from pyspark.sql.window import Window
```

Row_number

```
df.withColumn('rowCol',row_number().over(Window.orderBy('Item_Identifier'))).
display()
```

Rank And Dense_rank

```
df.withColumn('rank',rank().over(Window.orderBy(col('Item_Identifier').desc()
)))\
.withColumn('denseRank',dense_rank().over(Window.orderBy(col('Item_Identifier
').desc()))).display()
```

Cumulative Sum

```
df.withColumn('cumsum',sum('Item_MRP').over(Window.orderBy('Item_Type').rowsB
etween(Window.unboundedPreceding,Window.currentRow))).display()
```

{gives the sum till the current row}

```
df.withColumn('totalsum',sum('Item_MRP').over(Window.orderBy('Item_Type').row
sBetween(Window.unboundedPreceding,Window.unboundedFollowing))).display()
```

{gives the sum of all the data}

User Defined Functions (UDF)

STEP - 1 (create Python function)

```
def my_func(x):
    return x*x
```

STEP - 2 (convert into PySpark UDF)

```
my_udf = udf(my_func)
```

```
Example:
df.withColumn('mynewcol',my_udf('Item_MRP')).display()
```

DATA WRITING

CSV

```
df.write.format('csv')\
        .save('/FileStore/tables/CSV/data.csv')
```

Writing Modes

Append

```
df.write.format('csv')\
        .mode('append')\
        .save('/FileStore/tables/CSV/data.csv')
                    OR
df.write.format('csv')\
        .mode('append')\
        .option('path','/FileStore/tables/CSV/data.csv')\
        .save()
```

Overwrite

```
df.write.format('csv')\
.mode('overwrite')\
.option('path','/FileStore/tables/CSV/data.csv')\
.save()
```

Error

```
df.write.format('csv')\
.mode('error')\
.option('path','/FileStore/tables/CSV/data.csv')\
.save()
```

Ignore

```
df.write.format('csv')\
.mode('ignore')\
.option('path','/FileStore/tables/CSV/data.csv')\
.save()
```

Paraquet File Format

```
df.write.format('parquet')\
.mode('overwrite')\
.option('path','/FileStore/tables/CSV/data.csv')\
.save()
```