Case Study : Financial Regulation Domain: Banking

All financial institutions have been introduced to a new financial regulation that will consider the impact of possible future events when calculating their risk exposure. For this change, all financial trading services will have to source data from far more data sources including both trade and risk data, which means large amounts of data that have not historically been required for accounting.

The data is of trading index listings for each trading day.

Tasks:

To adhere to the regulation a leading trading service provider has decided to create a new repository (Big Data) and a multi-facet platform that can cater to these models. As part of the R&D team, you are required to execute a POC exercise to maintain a master data model.

1. Test the spark environment by executing the spark's HdfsTest.scala example.

2. Try to implement the same example in pyspark and perform spark-submit.

3. Analyze the behavior of spark application on Spark web UI

4. Edit the application and add custom logs. Once executed check the Spark logs.

5. Transfer the sample dataset from RDBMS to HDFS

6. Validate the loaded data by comparing the statistics of data both in source and HDFS

7. Create a new directory EQ in HDFS and transfer the data where series is EQ

8. Set total trades which are less than 500 to 0 and transfer only updated rows.

Solution:

Task 1 : Test the spark environment by executing the spark's HdfsTest.scala example

Location of HdfsTest.scala

/opt/cloudera/parcels/SPARK2-2.1.0.cloudera2-1.cdh5.7.0.p0.171658/lib/spark2/examples/src/main/scala/org/apache/spark/examples/HdfsTest.scala

Location of the Spark shell and the spark Submit Binary Files

/opt/cloudera/parcels/SPARK2-2.1.0.cloudera2-1.cdh5.7.0.p0.171658/lib/spark2/bin

After changing to bin directory , run

spark-shell

In spark -shell , then use the following command to copy paste the HdfsTest.scala

```
scala> :paste -raw
// Entering paste mode (ctrl-D to finish)

package org.apache.spark.examples

import org.apache.spark.sql.SparkSession


object HdfsTest {

  /** Usage: HdfsTest [file] */
  def main(args: Array[String]) {
    if (args.length < 1) {
      System.err.println("Usage: HdfsTest <file>")
      System.exit(1)
    }
    val spark = SparkSession
      .builder
      .appName("HdfsTest")
      .getOrCreate()
    val file = spark.read.text(args(0)).rdd
    val mapped = file.map(s => s.length).cache()
    for (iter <- 1 to 10) {
      val start = System.currentTimeMillis()
      for (x <- mapped) { x + 2 }
      val end = System.currentTimeMillis()
      println("Iteration " + iter + " took " + (end-start) + " ms")
    }
    spark.stop()
  }
}

// Exiting paste mode, now interpreting.
```

Then , ran the following command on the shell to execute the Scala Script.

```
scala> HdfsTest.main(Array("AppleStore.csv"))
Iteration 1 took 6142 ms
Iteration 2 took 79 ms
Iteration 3 took 41 ms
Iteration 4 took 51 ms
Iteration 5 took 27 ms
Iteration 6 took 26 ms
Iteration 7 took 30 ms
Iteration 8 took 26 ms
Iteration 9 took 25 ms
Iteration 10 took 26 ms
```

Task 2 : Try to implement the same example in pyspark and perform spark-submit.

Command Used :

```
./spark-submit                    /mnt/home/edureka_960126/HdfsTest.py
/user/edureka_960126/AppleStore.csv
```

HdfsTest.py

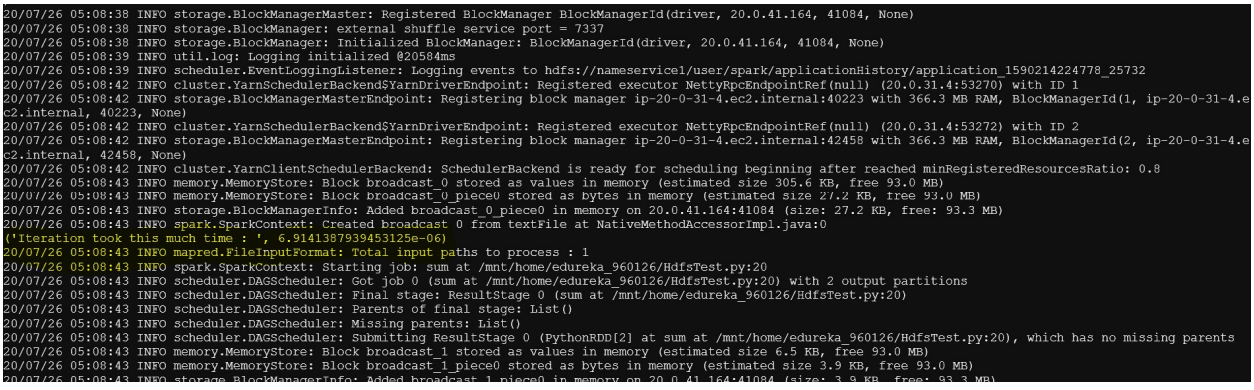
```
import sys
import os
import random
from operator import add, mul
from pyspark import SparkContext, SparkConf
from pyspark import SparkFiles
import time

def main(arg):
   file = arg
   conf = SparkConf().setAppName("HdfsTest")
   sc = SparkContext(conf=conf)
   rdd = sc.textFile(file).map(lambda line: len(line))

   for iter in range(10):
      start=time.time()
      rdd.map(lambda x : x+2)
      duration=time.time()-start
      print("Iteration took this much time : ", duration)
      print("Returned length(s) of: " , rdd.sum())


if __name__=="__main__":
   main(sys.argv[1])
```

Above is the file used to submit the job and the following results were obtained :



Task 3: Analyze the behavior of spark application on Spark web UI

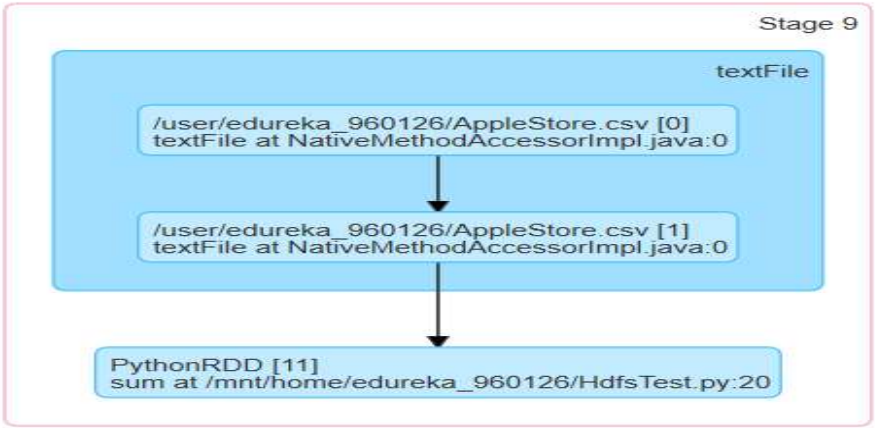
Completed Jobs (10)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
9	sum at /mnt/home/edureka_960126/HdfsTest.py:20	2020/07/26 05:08:48	40 ms	1/1	2/2
8	sum at /mnt/home/edureka_960126/HdfsTest.py:20	2020/07/26 05:08:48	43 ms	1/1	2/2
7	sum at /mnt/home/edureka_960126/HdfsTest.py:20	2020/07/26 05:08:48	51 ms	1/1	2/2
6	sum at /mnt/home/edureka_960126/HdfsTest.py:20	2020/07/26 05:08:48	78 ms	1/1	2/2
5	sum at /mnt/home/edureka_960126/HdfsTest.py:20	2020/07/26 05:08:48	80 ms	1/1	2/2
4	sum at /mnt/home/edureka_960126/HdfsTest.py:20	2020/07/26 05:08:48	82 ms	1/1	2/2
3	sum at /mnt/home/edureka_960126/HdfsTest.py:20	2020/07/26 05:08:48	56 ms	1/1	2/2
2	sum at /mnt/home/edureka_960126/HdfsTest.py:20	2020/07/26 05:08:48	91 ms	1/1	2/2
1	sum at /mnt/home/edureka_960126/HdfsTest.py:20	2020/07/26 05:08:47	0.1 s	1/1	2/2
0	sum at /mnt/home/edureka_960126/HdfsTest.py:20	2020/07/26 05:08:43	4 s	1/1	2/2

1. Application HdfsTest was there is the Spark History Server
2. Action was called 10 times which is sum() in 10 iterations as shown above which are shown above and were successful
3. Each time action was called the job was created.
4. Remember DAG, stages, tasks (created based on the transformations) are then distributed to worker nodes and then pulling results by the driver to display on the console.
   DAG shown below for stage 9 i.e. job 9

Total Time Across All Tasks: 30 ms
Locality Level Summary: Node local: 2
Input Size / Records: 64.0 KB / 7198

▾ DAG Visualization



Shown above are the transformations and the action (Note same as one shown DAG is created for each job)

Tasks (2)

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Input Size / Records	Errors
0	18	0	SUCCESS	NODE_LOCAL	2 / ip-20-0-31-4.ec2.internal stdout stderr	2020/07/26 05:08:48	16 ms		0.0 B / 3618	
1	19	0	SUCCESS	NODE_LOCAL	1 / ip-20-0-31-4.ec2.internal stdout stderr	2020/07/26 05:08:48	14 ms		64.0 KB / 3580	

Note that 2 Tasks were created for this , which were executed by 2 executors in the same node(worker) having hostname ip-20-0-31-4.ec2.internal

Task 4. Edit the application and add custom logs. Once executed check the Spark logs.

spark.eventLog.dir is the parameter to set the path where the logs are stored for the spark jobs by default.

Parameter set in our environment is :

hdfs://nameservice1/user/spark/applicationHistory

Using the below command we can see the logs on the console :

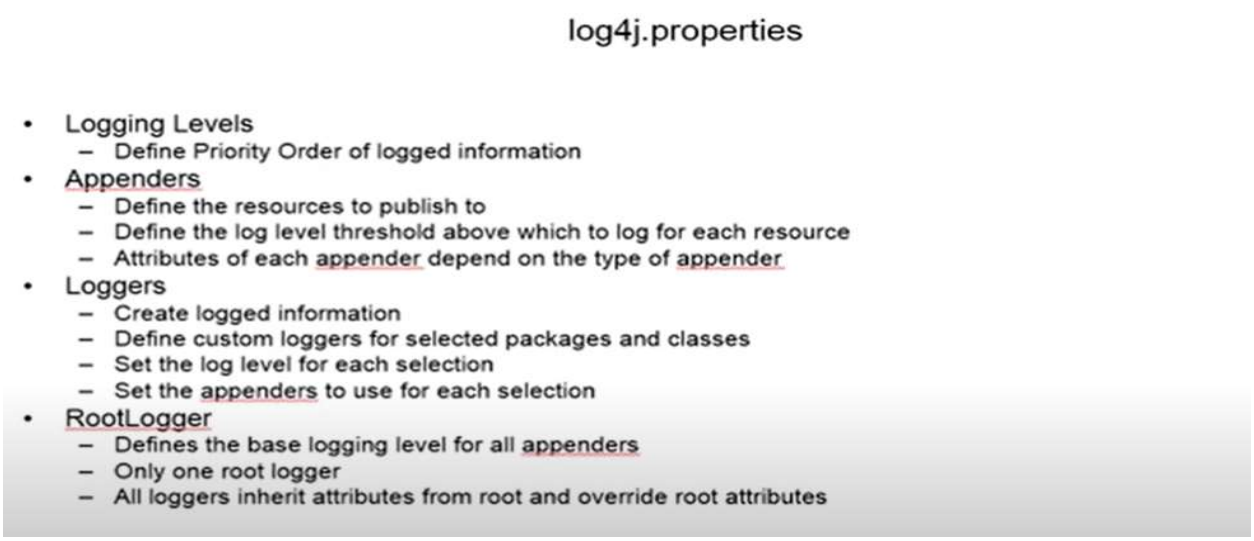
```
hdfs dfs -cat hdfs://nameservice1/user/spark/applicationHistory/application_1590214224778_25728
```

We can create the Custom logs for our application :

1. We can change the directory path where we want to store the logs instead of default directory path in the configuration:
   ./spark-submit --conf "spark.eventLog.dir=/mnt/home/edureka_960126/" /mnt/home/edureka_960126/HdfsTest.py /user/edureka_960126/AppleStore.csv
   This was verified and the logs were stored in the desired directory.

2. We can store the custom things in the log files :

Spark uses Log4j which is standard for Logging for various Java applications.

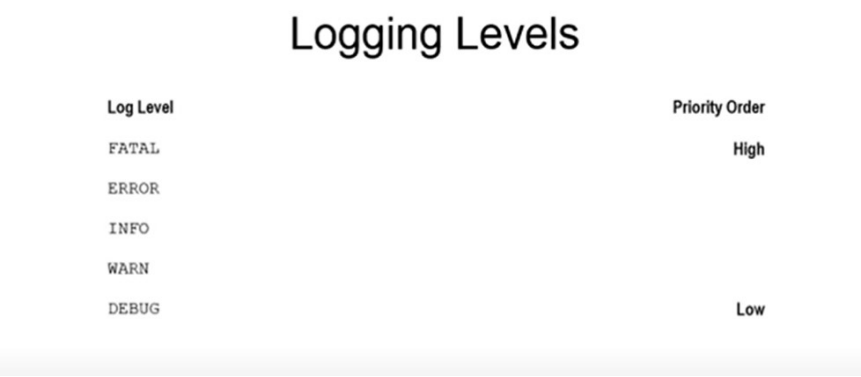


Logging Levels are the levels of essentiality we want logs to be collected.

Appenders are the one who publishes to various interfaces like files , e mails etc.

Loggers are the ones who collect the logs from the application.

Root Logger is the default and other Loggers inherit from the Root Logger.



Custom log4j.properties file was created and was submitted with spark submit for being used.

Command Used :

./spark-submit --conf "spark.driver.extraJavaOptions=-Dlog4j.debug=true" --conf "spark.driver.extraJavaOptions=-Dlog4j.configuration=log4j.properties" --files /mnt/home/edureka_960126/log4j.properties /mnt/home/edureka_960126/HdfsTest.py /user/edureka_960126/AppleStore.csv

Uploading resource file:/mnt/home/edureka_960126/log4j.properties -> hdfs://nameservice1/user/edureka_960126/.sparkStaging/application_1590214224778_26194/log4j.properties

Got such in the logs in the console which means that our log4j.properties was uploaded to the executors to be used.

5. **Transfer the sample dataset from RDBMS to HDFS**

a. Firstly , I loaded the file in local file system using FTP interface
b. Then loaded the CSV file to the HDFS using the copy from Local Command
   `hdfs dfs -copyFromLocal FINAL_FROM_DF.csv /user/edureka_960126/`
c. Then exported the Data from HDFS to RDBMS
   Used Sqoop Utility for this :

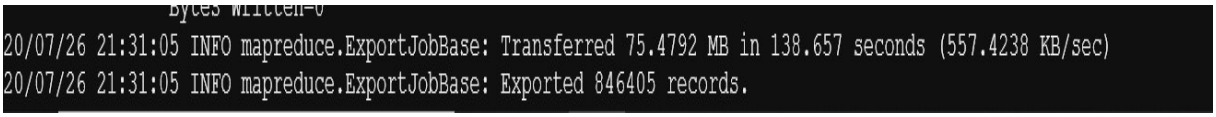
Table was created with the schema of the file :

Create table mod4_cs1 (SYMBOL Varchar(255) , SERIES Varchar(255) , OPEN Varchar(255),HIGH Varchar(255),LOW Varchar(255),CLOSE Varchar(255) , LAST Varchar(255) ,PREVCLOSE Varchar(255),

TOTTRDQTY Varchar(255), TOTTRDVAL Varchar(255),TIMESTAMP Varchar(255),TOTALTRADES Varchar(255), ISIN Varchar(255));

Then Data was exported from HDFS to RDBMS

sqoop export --connect jdbc:mysql://dbserver.edu.cloudlab.com/labuser_database --username edu_labuser --password edureka --table mod4_cs1 --export-dir /user/edureka_960126/mod4_cs1_data



d. Then imported the Data from RDBMS to HDFS as desired.

- First we viewed the table in the RDBMS:

Mod4_cs1 is the Table Name
Hdfs directory name where the Data was stored :
/user/edureka_960126/mod4_cs1_data

Select * from mod4_cs1 ;

- Then exported data from the RDBMS to HDFS again as assigned in the task :

First Primary Key was added in order to import from RDBMS :

```
ALTER TABLE mod4_cs1  ADD `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY
```

```
Hence an autoincrement column was added
```

sqoop import --connect jdbc:mysql://dbserver.edu.cloudlab.com/labuser_database --username edu_labuser --password edureka --table mod4_cs1 --target-dir /user/edureka_960126/cs1_mod4_data

6. Validate the loaded data by comparing the statistics of data both in source and HDFS

Statistics are validated as the number of rows in MYSQL Table and the HDFS records in the directory are same.

7. **Create a new directory EQ in HDFS and transfer the data where series is EQ**

Command Used :

sqoop import --connect jdbc:mysql://dbserver.edu.cloudlab.com/labuser_database --username edu_labuser --password edureka --table mod4_cs1 --target-dir /user/edureka_960126/cs1_mod4_eq --where "SERIES = 'EQ'"

There are 73199 records in the table where SERIES=EQ

8. **Set total trades which are less than 500 to 0 and transfer only updated rows.**

- Sqoop provides an incremental import mode which can be used to retrieve only rows newer than some previously-imported set of rows.
- Sqoop supports two types of incremental imports:
- append and lastmodified.
- You can use the –incremental argument to specify the type of incremental import to perform.

First changed the timestamp column with data type time stamp.

```
ALTER TABLE mod4_cs1  MODIFY COLUMN TIMESTAMP timestamp ;
```

Sqoop supports two types of incremental imports: `append` and `lastmodified`.

You should specify `append` mode when importing a table where new rows are continually being added with increasing row id values. You specify the column containing the row's id with `--check-column`. Sqoop imports rows where the check column has a value greater than the one specified with `--last-value`.

An alternate table update strategy supported by Sqoop is called `lastmodified` mode. You should use this when rows of the source table may be updated, and each such update will set the value of a last-modified column to the current timestamp. Rows where the check column holds a timestamp more recent than the timestamp specified with `--last-value` are imported.

At the end of an incremental import, the value which should be specified as `--last-value` for a subsequent import is printed to the screen. When running a subsequent import, you should specify `--last-value` in this way to ensure you import only the new or updated data. This is handled automatically by creating an incremental import as a saved job, which is the preferred mechanism for performing a recurring incremental import. See the section on saved jobs later in this document for more information

So in our case we have to perform the incremental import.

Without performing the updation the sqoop command was run for experimental purpose :

Command used :

sqoop import --connect jdbc:mysql://dbserver.edu.cloudlab.com/labuser_database --check-column TIMESTAMP --username edu_labuser --password edureka --table mod4_cs1 --target-dir /user/edureka_960126/cs1_mod4_updated --incremental lastmodified --last-value 2017-06-28

Information which we got in return of job execution :

SELECT MIN(`id`), MAX(`id`) FROM `mod4_cs1` WHERE (`TIMESTAMP` >= '2017-06-28' AND `TIMESTAMP` < '2020-07-27 01:59:43.0')

The above command was actually executed when we executed the above sqoop command to import the records from HDFS.

20/07/27 02:00:29
INFO tool.ImportTool:
--incremental lastmodified
INFO tool.ImportTool:
--check-column TIMESTAMP
INFO tool.ImportTool:
--last-value 2020-07-27 01:59:43.0

```
MySQL [labuser_database]> SELECT COUNT(`id`) FROM `mod4_cs1` WHERE ( `TIMESTAMP` >= '2017-06-28' AND `TIMESTAMP` < '2020-07-27 01:59:43.0' )
    -> ;
+-------------+
| COUNT(`id`) |
+-------------+
|      228259 |
+-------------+
1 row in set (0.40 sec)

MySQL [labuser_database]> SELECT COUNT(`id`) FROM `mod4_cs1`;
+-------------+
| COUNT(`id`) |
+-------------+
|      846404 |
+-------------+
1 row in set (0.21 sec)

MySQL [labuser_database]> SELECT MAX(`id`) FROM `mod4_cs1`;
+-----------+
| MAX(`id`) |
+-----------+
|    846405 |
+-----------+
1 row in set (0.00 sec)
```

As we have seen in the SQOOP Records also , 228259 records were taken into the example folder which have last modified as >= 2017-06-28 to present.

And we got certain parameters in return which can be used for making again SQOOP import.

Performing the task in hand

1. Set total trades which are less than 500 to 0
2. and transfer only updated rows.

For 1:

SQL Query we ran :

```
MySQL [labuser_database]> SELECT COUNT(`id`) FROM `mod4_cs1` where TOTALTRADES < 500 ;
+-------------+
| COUNT(`id`) |
+-------------+
|      399261 |
+-------------+
1 row in set (0.30 sec)
```

So we checked the number of rows that will be updated to confirm that our SQOOP and update column is working in sync.

Update table mod4_cs1 TIMESTAMP=CURRENT_TIMESTAMP() , TOTALTRADES=0 WHERE TOTALTRADES < 500 ;

```
MySQL [labuser_database]> Update mod4_cs1 set TIMESTAMP=CURRENT_TIMESTAMP() , TOTALTRADES=0 WHERE TOTALTRADES < 500 ;
Query OK, 399261 rows affected (45.03 sec)
Rows matched: 399261  Changed: 399261  Warnings: 0
```

Sqoop to get only updated ones based on the timestamp :

sqoop import --connect jdbc:mysql://dbserver.edu.cloudlab.com/labuser_database --check-column TIMESTAMP --username edu_labuser --password edureka --table mod4_cs1 --target-dir /user/edureka_960126/cs1_mod4_updated_ttrades --incremental lastmodified --last-value 2020-07-27 01:59:43.0

```
        Total megabyte-milliseconds taken by all map tasks=509
  Map-Reduce Framework
          Map input records=399261
          Map output records=399261
          Input split bytes=428
          Spilled Records=0
          Failed Shuffles=0
          Merged Map outputs=0
          GC time elapsed (ms)=796
          CPU time spent (ms)=16260
```

So same number of records that were updated : 399261

```
20/07/27 02:37:02 INFO tool.ImportTool:  --incremental lastmodified
20/07/27 02:37:02 INFO tool.ImportTool:   --check-column TIMESTAMP
20/07/27 02:37:02 INFO tool.ImportTool:   --last-value 2020-07-27 02:36:27.0
20/07/27 02:37:02 INFO tool.ImportTool: (Consider saving this with 'sqoop job --create')
```

----------NOT PART OF ASSIGNMENT----------
----------SPARK NOTES MADE AFTER LEARNING WHICH WERE USED IN PERFORMING THE ASSIGNMENT----------

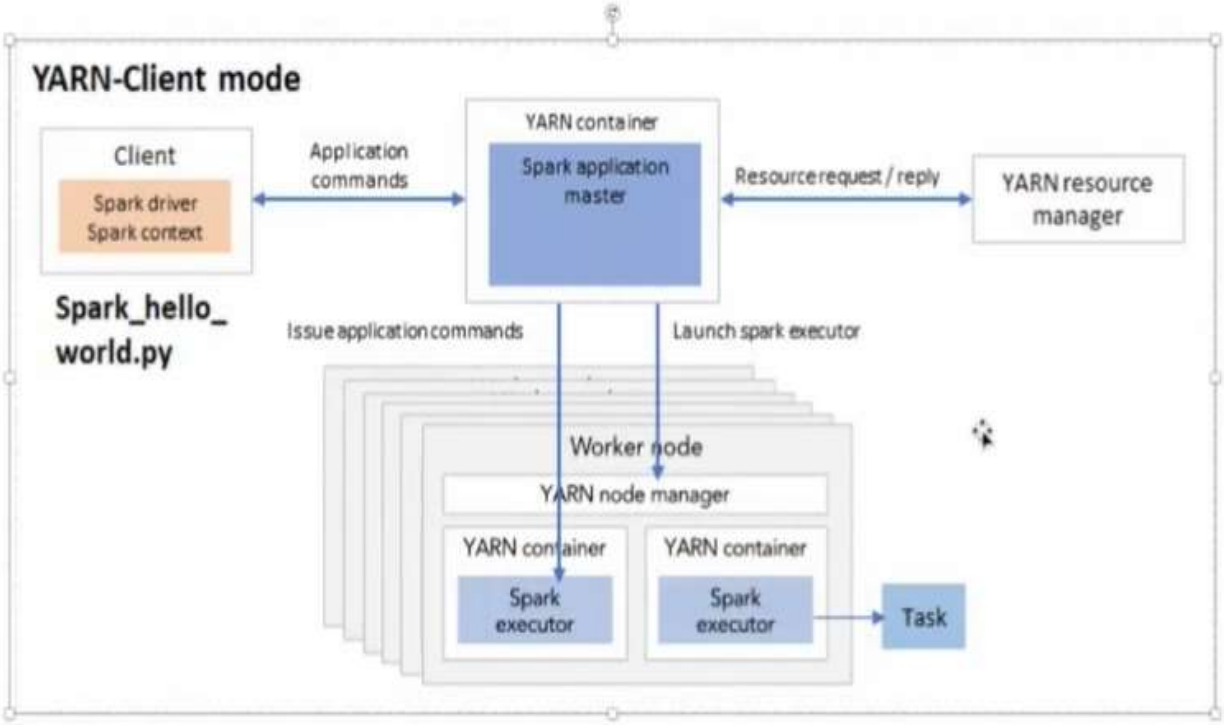
Brief Theory about Spark is presented below:

Spark is deployed in two modes :

1. **Client Mode**
2. **Cluster Mode**

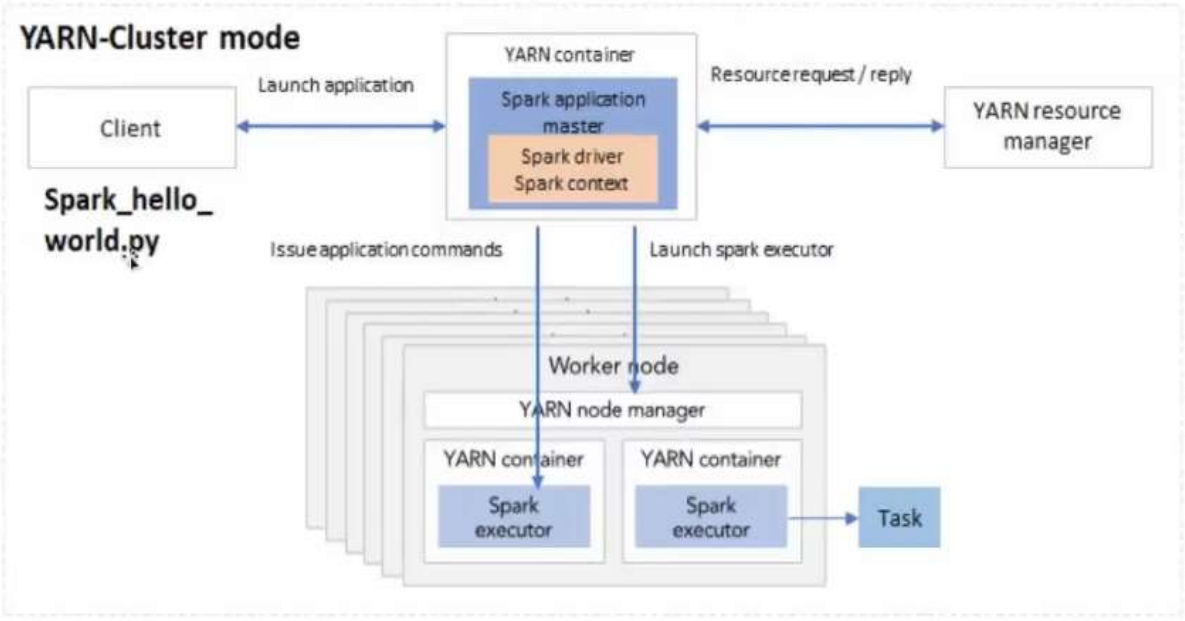
Sequence of Steps in Client Mode :-

1. When you open the Spark Shell , then SparkSession as spark and SparkContext as sc are by default available. These are created by Spark Driver present in your machine.
2. This Spark Context is used to send the application commands to the Spark Application Master running in the cluster which in turn will negotiate for resources with YARN resource Manager.(Note that this Spark Application Master is just a program launched in a yarn container (a JVM machine))
3. Based on negotiation , our application Master will launch executors which are JVM only through the YARN Node Manager which are present there in the worker nodes (or just nodes in the Yarn Cluster)



4. Then further issue commands to those Spark Executors.

Sequence of Steps in Cluster Mode :-

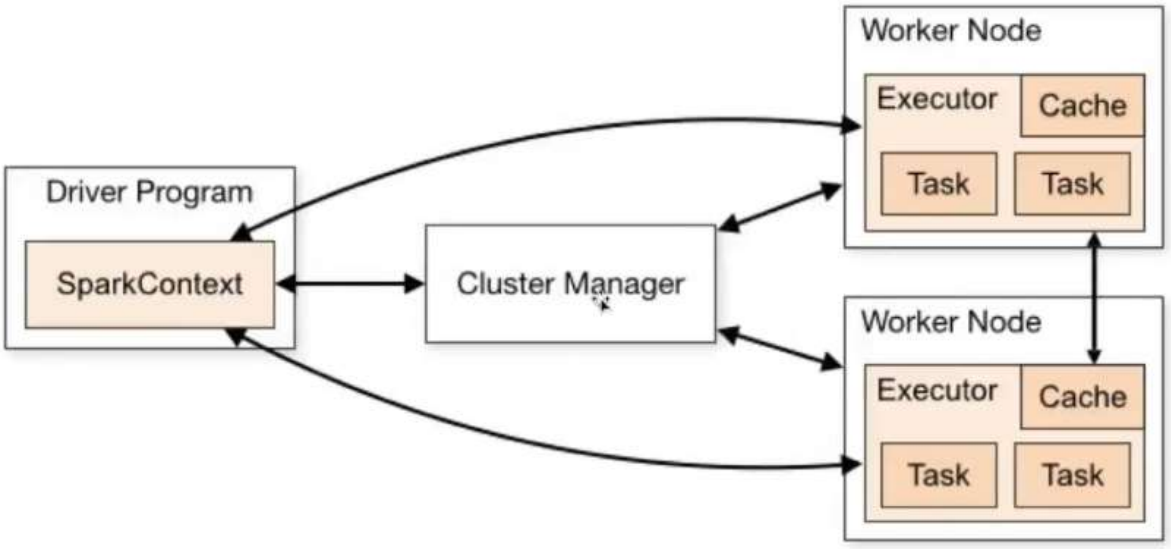


1. Client Launch application
2. Assuming here the YARN Cluster , in one of the Yarn Container in one of the nodes of the cluster , Spark Application Master gets created(or may be already running so connection between the client and the spark application Master is there)
3. Spark Application Master itself create the SparkContext or Session and Spark Driver Program further negotiate for resources with Yarn resource Manager to launch the executors.

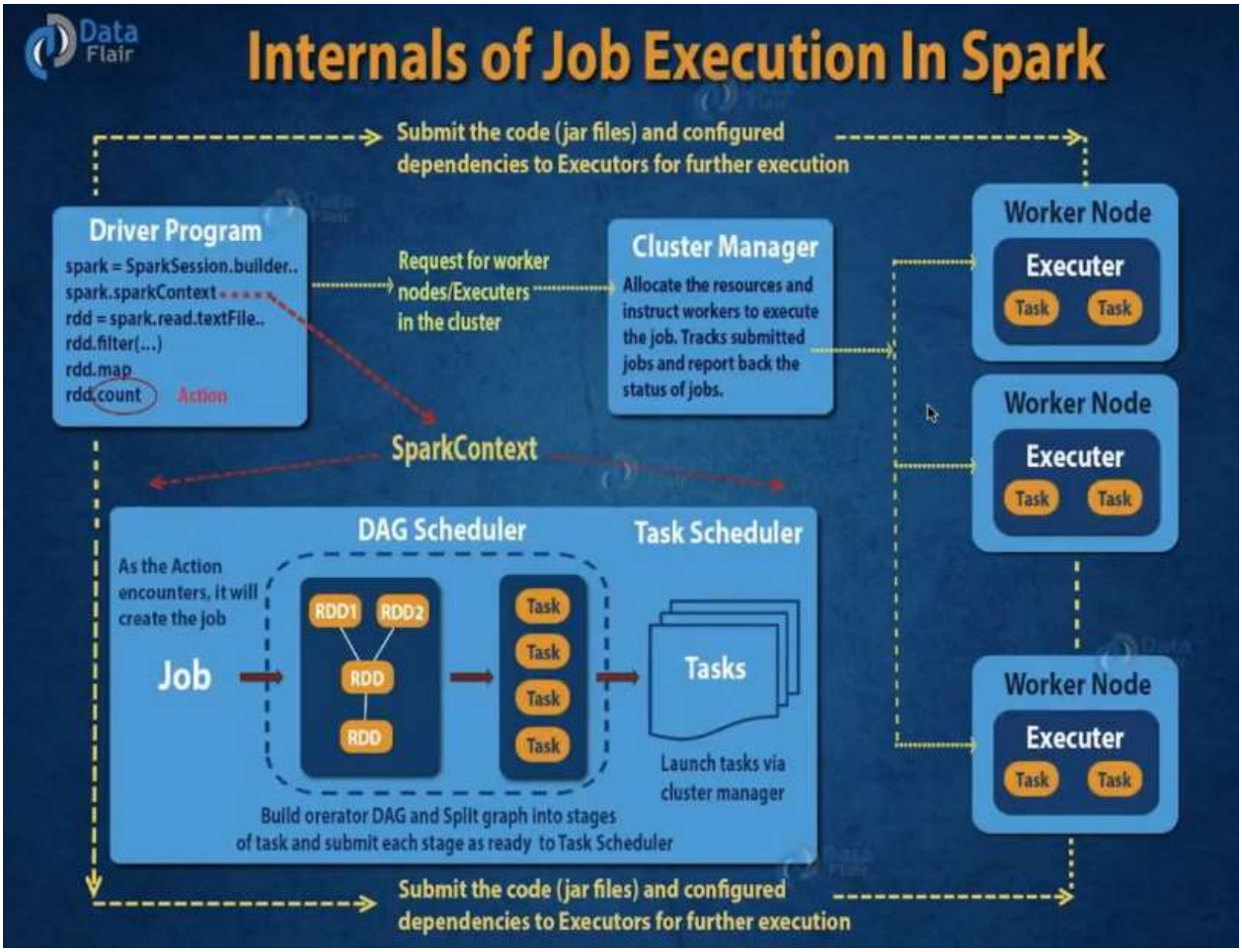
When we submit our job through the Spark Shell then we use the Client Mode .

General Steps in case of Spark execution are :-

1. Spark Driver creates a Spark Context
2. Spark Context establishment is the establishment of connection with the Cluster Manager
3. Driver will ask for resources to the Cluster Manager and create executors in containers available in the other nodes in the cluster which are hereby called worker nodes.
4. So, the Driver provides a task (code/compute + Data) to Cluster manager(here Yarn) , Yarn decides decide where(i.e. in which worker node /executor) Data resides and in turn will allocate memory etc. resources to the executor to execute the task.
5. Note that the Driver and Yarn has an intermediary of the **Application Master and hence Driver communicate with the Yarn through the Application Master.**



Above is the simplistic diagram showing the Driver/Leader how it first creates SparkContext , then communicate with Cluster Manger through Spark Application Master and then get the executors launched which in turn execute tasks assigned by the driver.



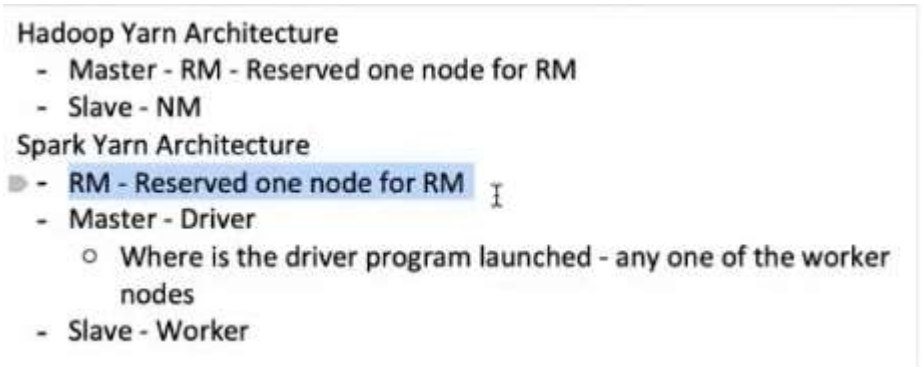
Description of the Diagram :

1. When Action is performed then the Driver Program request for Worker Nodes/Executors through the Cluster Manager.
2. Whatever program/spark application we will write will constitute a driver program .
3. As it can be seen that when action is encountered the DAG is created , tasks division etc. take place (number of tasks depend on the number of available resources) and then code files are being submitted to executors through Yarn(i.e. Code / Driver Program / Jar Files) to the Executors for executing the tasks

Spark Benefits :

In memory Data sharing against Hadoop storing intermediate results using HDFS and then shuffling.(so avoid lot of disk input and output)

Yarn Vs. Spark on Yarn Architecture



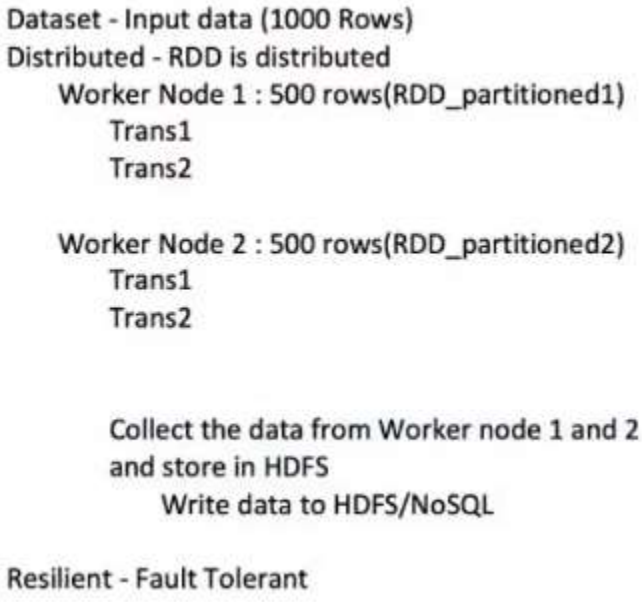
Yarn has Master Resource Managers and individual Nodes which have Nodes Manager.

Spark On Yarn has Spark Driver running on any Yarn Node and then through Yarn Master/RM ask for resources and hence executing tasks on Yarn worker nodes.

Important Spark Concepts :

1. RDDs
2. DAG
3. Lazy Evaluation

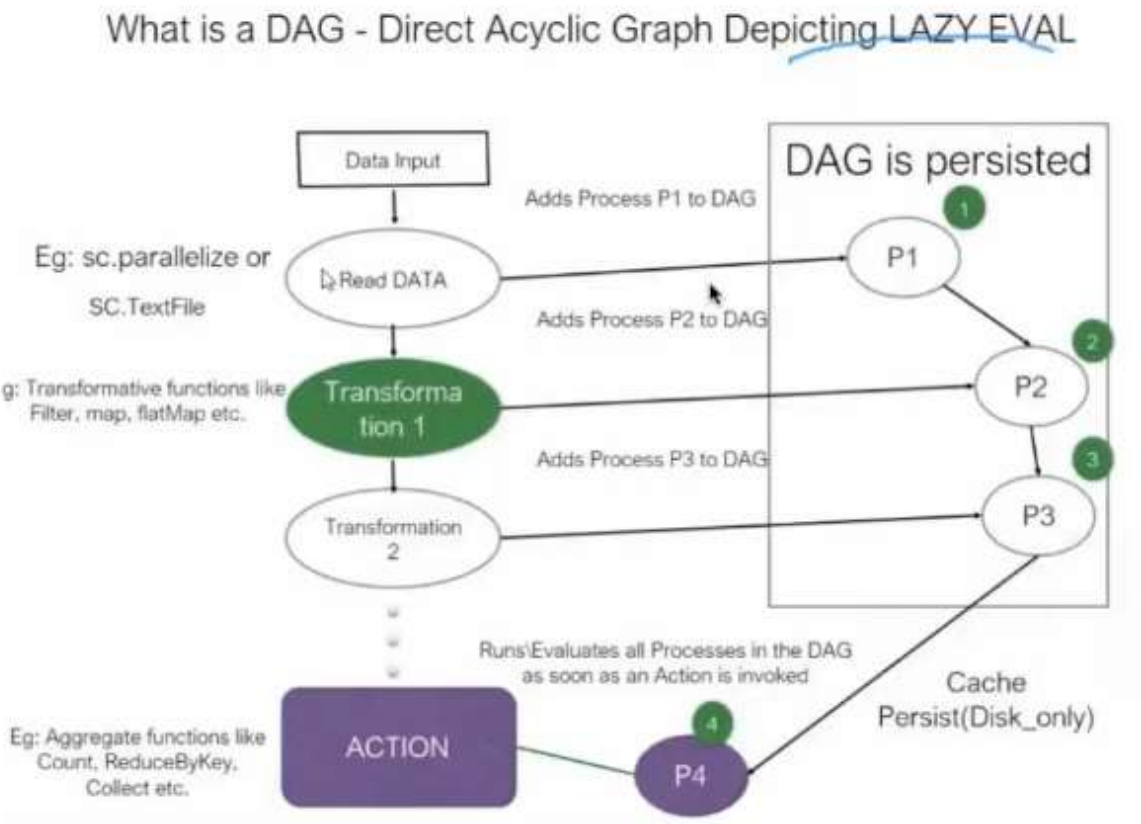
RDDs

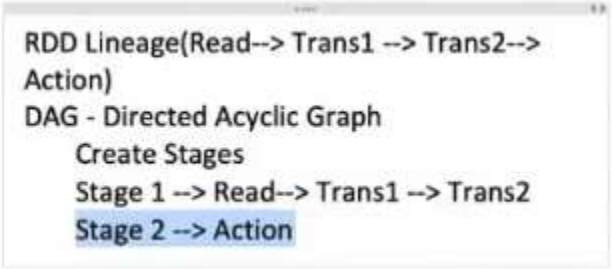


Dataset is what is input by us.

Distributed as when we transform our input data into the RDD then our Data gets distributed over the worker nodes.

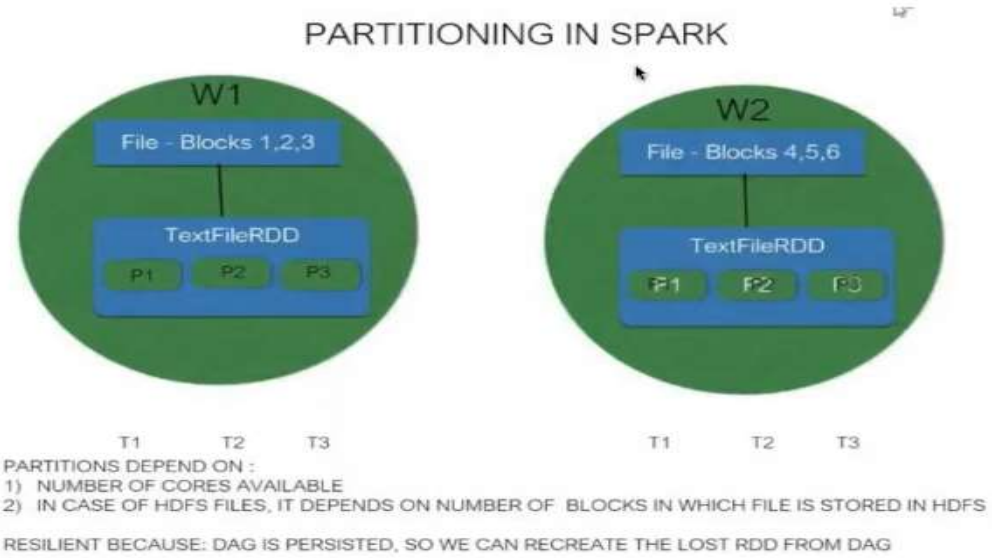
Resilient – due to fault tolerance way of replication of partitions





Important point :

- In Memory Data Sharing happens
- Action will not just lead to data collected at Driver but in any of the worker also like then saving all data collected to HDFS/NoSQL etc.
- Driver is responsible only for storing the DAG and workers only have to do any transformations and actions.
- When you read Data it creates RDD and when you perform Trans1 then RDD and then Transform 2 then RDD so on. So the concept of RDD was introduced in Spark.



Why RDD and how the Spark provides you with Fault tolerance ?

9. Persistence of DAG , which makes possible to move to another node in case of Node Failure and again use DAG to perform the failed tasks and recreate RDDs doing transformation on previous immutable RDDs on another Node.
10. DAG keeps the Lineage of RDDs in case of failure happens it(Spark) will recreate an RDD
11. Replication of Distributed partitions of Datasets.

Note RDDs are immutable. As we know that RDDs are created at every stage of reading , transformation etc. and are important to be immutable

Difference between the ReduceByKey and GroupByKey