Lab -10

1. **Aim :** Learn and apply the architecture of Apache Spark for data analytics. Solve Word Count program requirement using Apache Spark.

2. **Objective** : Students will learn various data cleaning, data transformation, data reductions techniques. Apache Spark is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters. Learn the architecture and working of Spark. Utilize the language of your choice to perform sample analytics. Note that when chosen HDFS as data storage behind the spark framework, integration of software components are learnt implicitly.

3. **Description:**

Spark is a robust and evolving engine for data anylytics. Spark works with data using RDDs (Resilient Distributed Dataset) and data frames. The operations are immutable. The api allowed to process the data uses an approach of lazy processing. This allows spark to have various workflows checked out and optimize the processing. It uses graphs to represent the dependencies and operations, which are also utilized while optimizing.

4. **Methodology:**

There are two ways to run a program in apache spark

1. Using Spark REPL
2. By submitting a jar file with spark-submit

1. Spark shell

Spark's shell provides a simple way to learn the API, as well as a powerful tool to analyze data interactively. It is available in either Scala (which runs on the Java VM and is thus a good way to use existing Java libraries) or Python.

- **Starting spark shell**

  We can start a spark-shell by using spark-shell <deployment mode>

- **Spark deployment mode**
  I. Local mode

  II. Standalone mode

  III. Cluster mode(YARN)

❖ **Working with Spark in Local mode**

A. Running in local mode with maximum core as possible
spark-shell - -master “local[*]”

B. Running in local mode with 4 cores
spark-shell --master "local[4]"-

```
[hadoop@hadoop-clone bin]$ spark-shell --master "local[4]"
23/09/22 16:18:29 WARN Utils: Your hostname, hadoop-clone resolves to a loopback address: 127.0.0.1; usi
ng 192.168.28.101 instead (on interface eno1)
23/09/22 16:18:29 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
23/09/22 16:18:33 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using
 builtin-java classes where applicable
Spark context Web UI available at http://192.168.28.101:4040
Spark context available as 'sc' (master = local[4], app id = local-1695379714022).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.5.0
      /_/

Using Scala version 2.12.18 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_192)
Type in expressions to have them evaluated.
Type :help for more information.
```

Note :
Once the shell is started the context object is available as sc.

❖ **Wordcount program Using Spark shell REPL**

1. **Create an RDD for inputfile**

```
scala> val inputfile=sc.textFile("file:///home/hadoop/Desktop/inputdata.txt")
;
inputfile: org.apache.spark.rdd.RDD[String] = file:///home/hadoop/Desktop/inp
utdata.txt MapPartitionsRDD[7] at textFile at <console>:23
```

Note :- sc.textfile("hdfs:/") for taking input from HDFS

2. **Display the data of the input file(optional)**

```
scala> inputfile.foreach(f=>{println(f)})
deer bear river
bear bear river
apple dear apple
bear river
dear apple apple cat
dear dear
cat
```

- **Count no of lines**

```
scala> inputfile.count
res16: Long = 5
```

- **Collect the output**

```
scala> inputfile.collect()
res17: Array[String] = Array(Hello World of Hadoop, hello hi, hello begin, begin enf, end of the end)
```

3. **Create a mapper and reducer**

```
scala> val counts=inputfile.flatMap(line=>line.split(" ")).map(word=>(word,1)
).reduceByKey(_ + _ );
counts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[10] at reduceBy
Key at <console>:23
```

- **RDD.toDebugString()**

  A description of this RDD and its recursive dependencies for debugging.

```
scala> counts.toDebugString
res6: String =
(2) ShuffledRDD[10] at reduceByKey at <console>:23 []
 +-(2) MapPartitionsRDD[9] at map at <console>:23 []
    |  MapPartitionsRDD[8] at flatMap at <console>:23 []
    |  file:///home/hadoop/Desktop/inputdata.txt MapPartitionsRDD[7] at textF
ile at <console>:23 []
    |  file:///home/hadoop/Desktop/inputdata.txt HadoopRDD[6] at textFile at
<console>:23 []
```

- ## What is RDD Persistence and caching

Spark RDD persistence is an optimization technique in which saves the result of RDD evaluation. Using this we s**ave the intermediate result** so that we can use it further if required. It reduces the computation overhead.
cache() and persist()

RDD.cache()
Persist this RDD with the default storage level (MEMORY_ONLY).

```
scala> counts.cache()
res7: counts.type = ShuffledRDD[10] at reduceByKey at <console>:23
```

1. MEMORY_ONLY
2. MEMORY_AND_DISK
3. MEMORY_ONLY_SER
4. MEMORY_AND_DISK_SER
5. DISK_ONLY

- **How to Unpersist RDD in Spark?**

  Spark monitors the cache of each node automatically and drop out the old data partition in the **LRU (least recently used)** fashion. LRU is an algorithm which ensures the least frequently used data. It spills out that data from the cache. We can also remove the cache manually using **RDD.unpersist()** method.

4. **Action - perform actions on created RDD**

- **collect -**

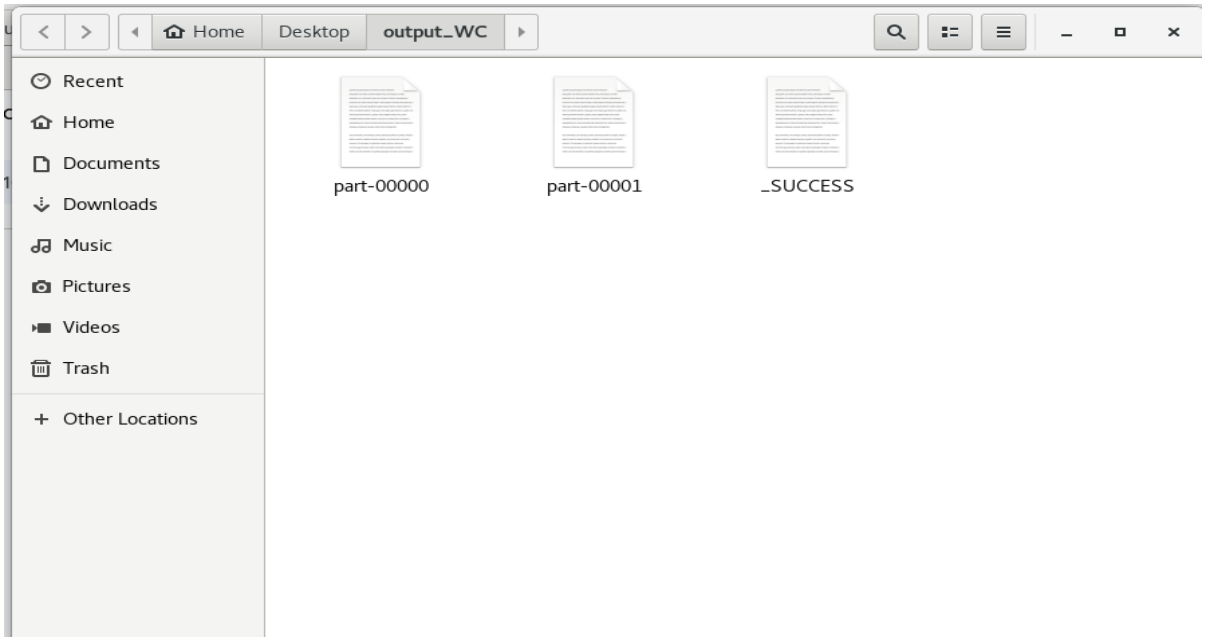
  Counts.collect will collect the data and display it on console

```
scala> counts.collect
res8: Array[(String, Int)] = Array((bear,4), (deer,1), (river,3), (apple,4),
(dear,4), (cat,2))
```

- **If you want to save your data as output file we can use the following command**

  counts.saveAsTextFile("file:///home/hadoop/Desktop/output_WC");

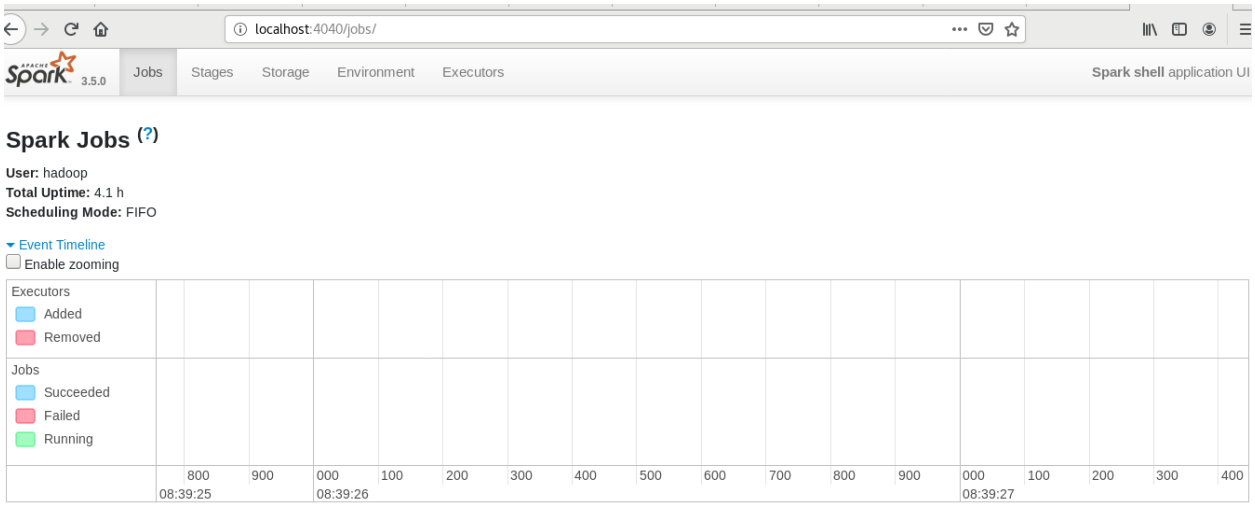
Output of reducer as shown below



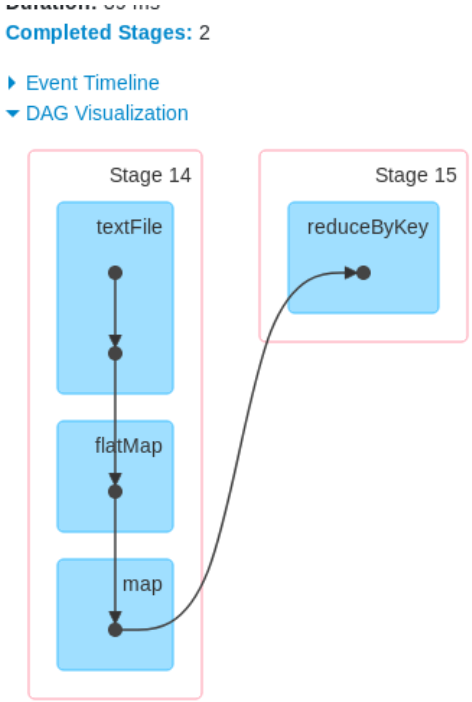
- **Observing DAG visualization**

  Goto url localhost:4040(default url for spark-shell)

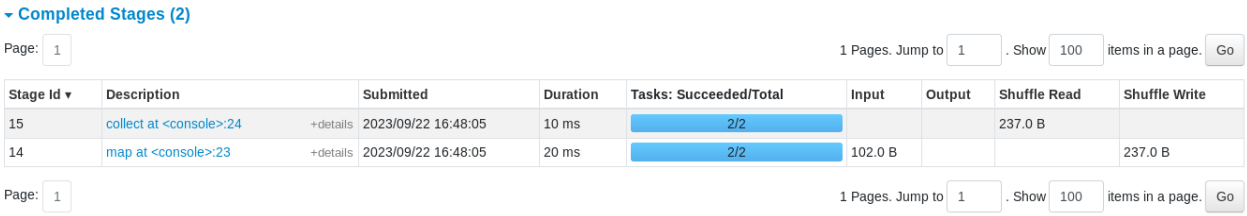
  It will display the following UI , we can analyze all jobs and their stages from here



  Goto **timeline of events** and explore **DAG visualization** for the submitted job



We can also observe the status of the jobs



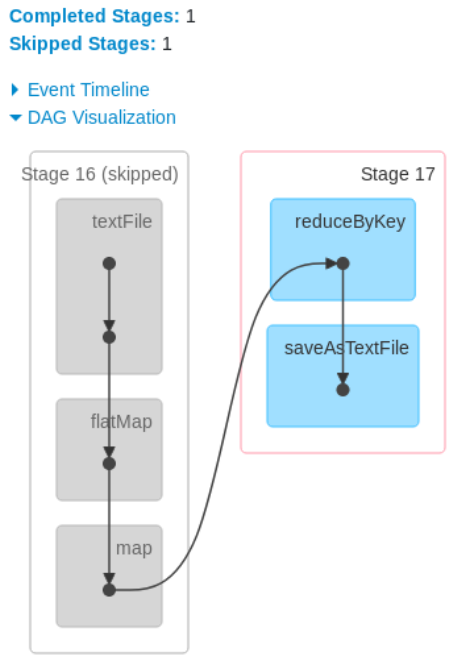
- # Wordcount Program by taking input from HDFS

1. **Create an input file in HDFS and upload it in spark as shown below**

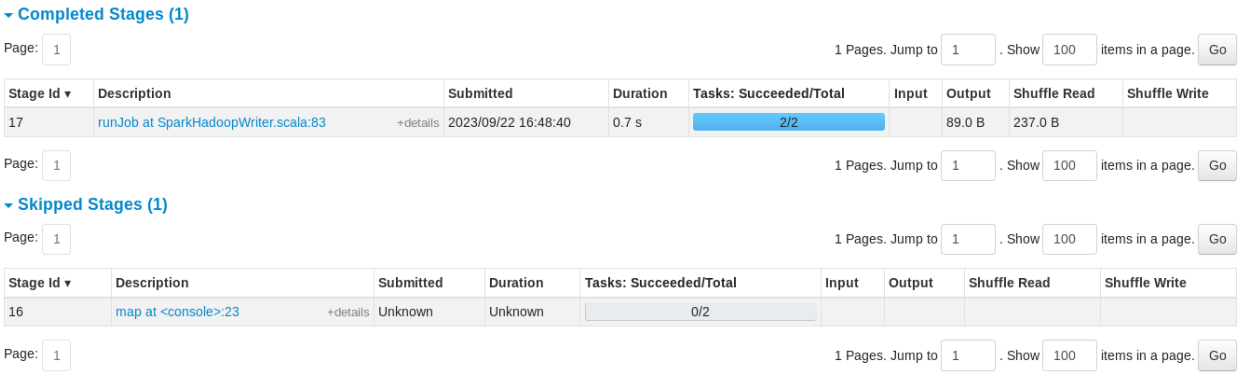
```
scala> val inputfile=sc.textFile("/wordcountdemo/input/file1.txt");
inputfile: org.apache.spark.rdd.RDD[String] = /wordcountdemo/input/file1.txt
MapPartitionsRDD[15] at textFile at <console>:23

scala> inputfile.foreach(f=>{println(f)})
begin enf
Hello World of Hadoop
end of the end
hello hi
hello begin
```

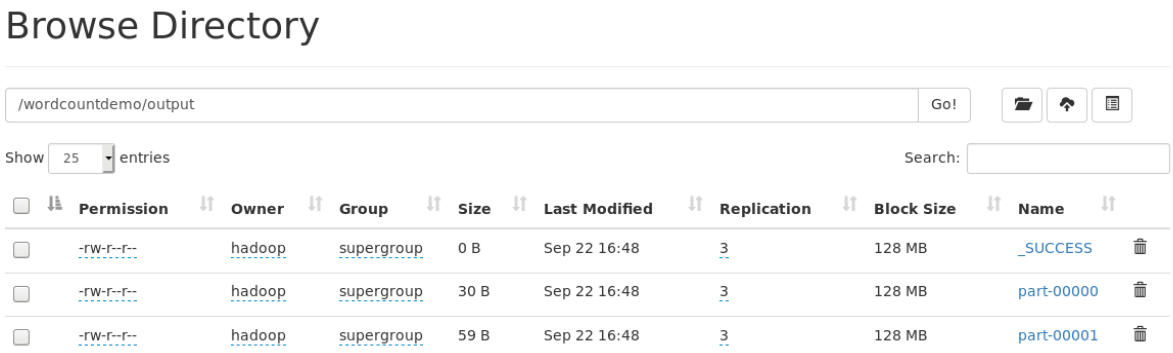
2. **Write the Map reduce steps**
3. **Observe the DAG visualization**



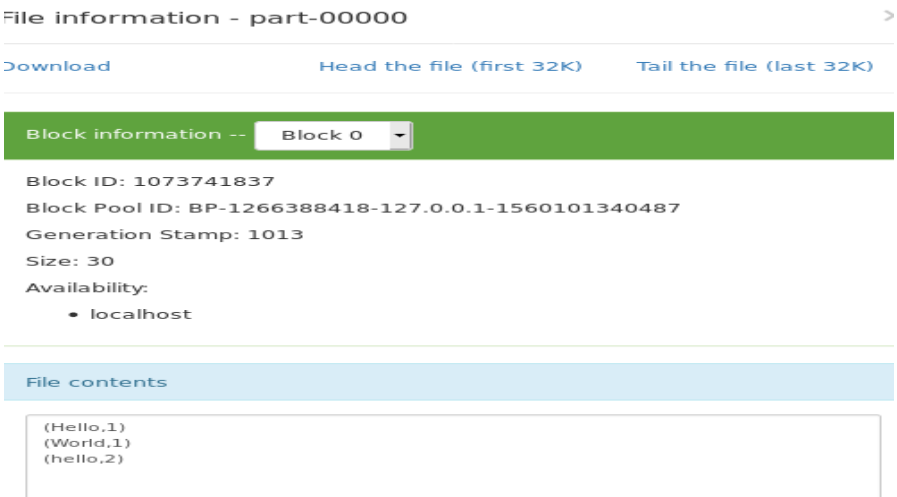
- **Output of submitted jobs**

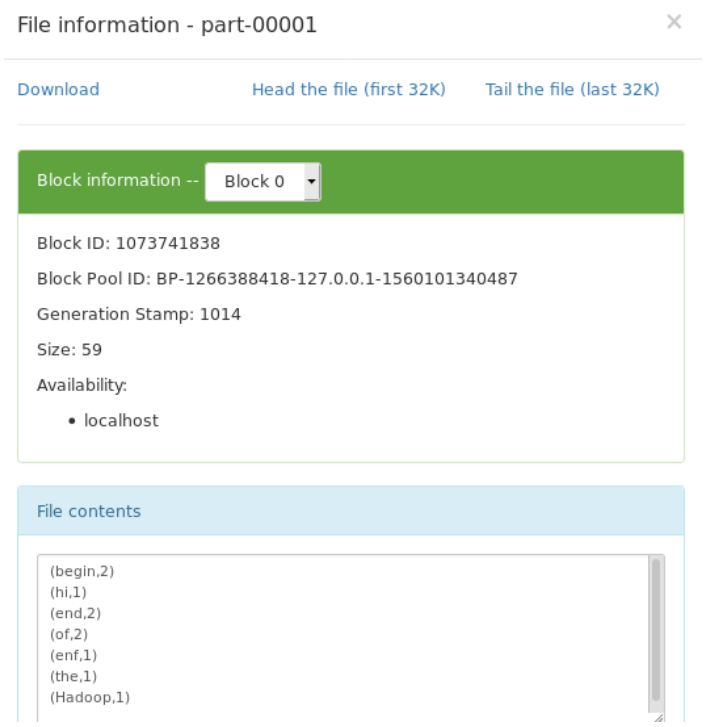


4. **Output of mapreduce can be shown in HDFS as below**



- **Part -0 , part-1**





Skipped stage

it means that data has been fetched from cache and there was no need to re-execute given stage. It is consistent with your DAG which shows that the next stage requires shuffling (reduceByKey). **Whenever there is shuffling involved Spark automatically caches generated data:**

Spark Standalone mode

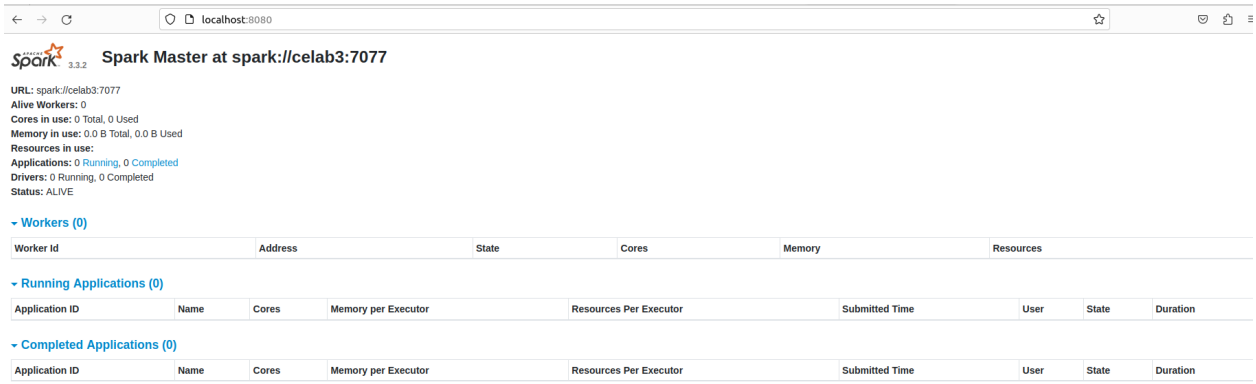
you can launch a standalone cluster either manually, by starting a master and workers by hand, or use our provided launch scripts. It is also possible to run these daemons on a single machine for testing.

- **Start Spark master and monitor in the browser**

  cd /opt/spark/sbin/start-master.sh

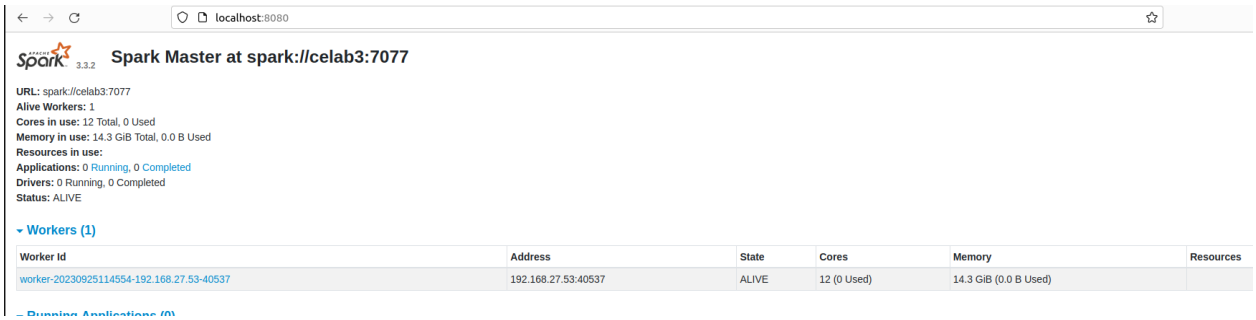
  http://locahost:8080

Spark-master UI



- **Start Spark worker and attach to above master. Monitor in the browser about registered worker with the master.**

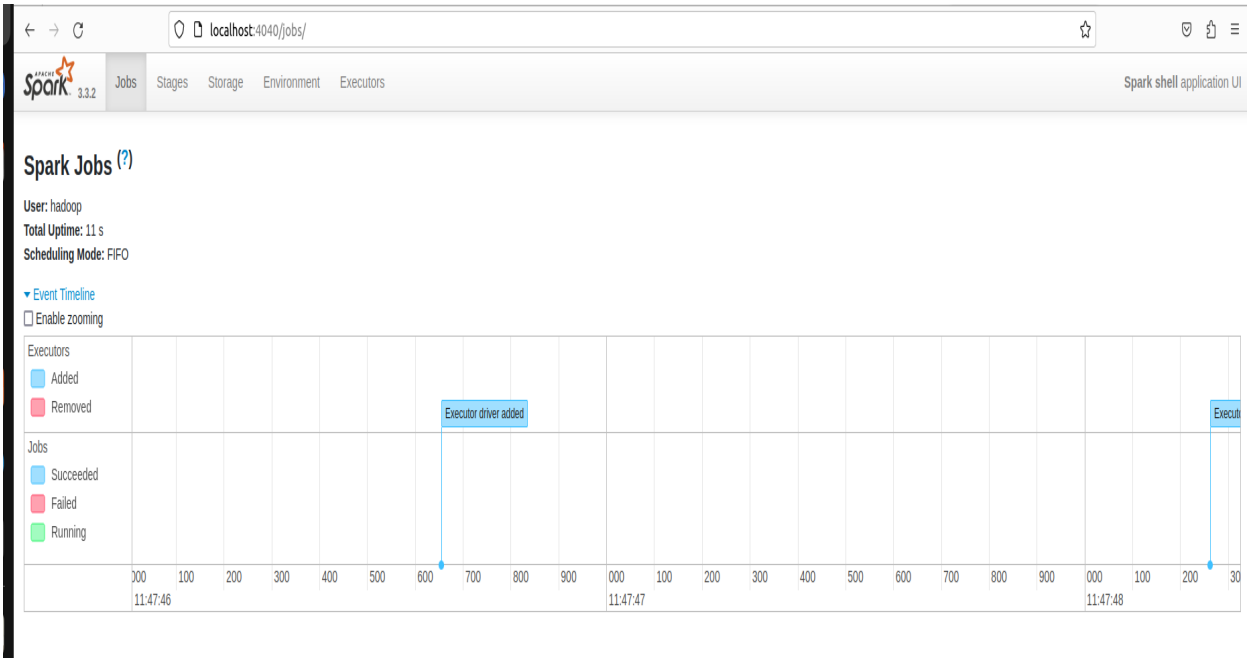
  sbin/start-worker.sh spark://localhost:7077
  http://locahost:8080



- **Start spark-shell on the master node**

  spark-shell --master spark://celab3:7077
  Where master url =spark://celab3:7077

  Once the shell is started the following GUI is available



There are 2 ways to execute the program

1) we can use IDE and create a jar file and then submit the jar file to the master for execution

I. **Now create a jar file in IDE or CLI**
II. **Run the jar file and check the output status**
III. **Submit the application to spark so that spark can execute the application**

spark-submit --master spark://localhost:7077 --deploy-mode=cluster --class=me.my.mine.App target/SparkWordCount-1.0-SNAPSHOT.jar

Refer the documentation link-1

2) use CLI to create a scala program and then run it directly on the shell
Refer the section of creating scala script

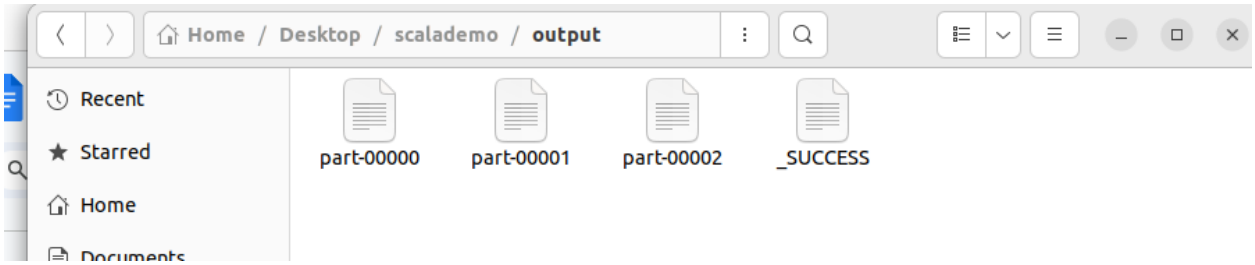
I. **Load the scala script**
In this approach we first laid the scala script using :load command.

```
scala> :load /home/hadoop/Desktop/scalademo/WC.scala
Loading /home/hadoop/Desktop/scalademo/WC.scala...
import org.apache.spark.sql.SQLContext
import org.apache.spark.{SparkConf, SparkContext}
defined object WC
```

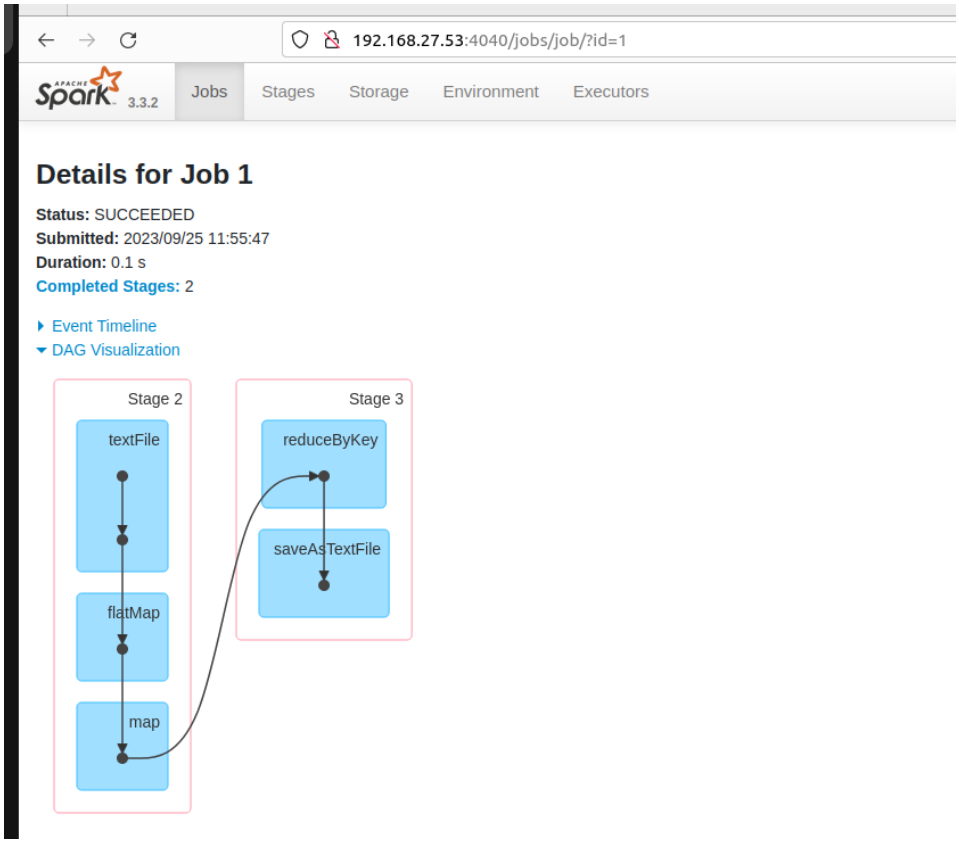
II. **Run the script by calling the main method of the WC object.**

```
cala> WC.main(Array("file:///home/hadoop/Desktop/scalademo/input.txt"))
```

III. **Output directory is created as mentioned in the code**



DAG Visualization



- **Spark Cluster Mode(Yarn Mode)**

```
spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
```

Word Count program using scala with Spark

src/main/scala/me/my/mine/App.scala

```
package me.my.mine
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
object App
{
	// Main Method
	def main(args: Array[String])
	{

	val conf = new
SparkConf().setAppName("App").setMaster("spark://localhost:7077")
	val sc = new SparkContext(conf)
	val text_file =
sc.textFile("hdfs://localhost:9000/user/sparkuser/my-dev/mapreduce/WordCount/input")
	text_file.collect.foreach(println)
	val counts = text_file.flatMap(line => line.split("
")).map(word=>(word,1)).reduceByKey((a,b)=>a+b).sortByKey()
	counts.collect.foreach(println)

counts.saveAsTextFile("hdfs://localhost:9000/user/sparkuser/my-dev/mapreduce/WordCo
unt/spark_output")
	}
}
```

Implementation Notes:

###Spark setup steps

```
chown -R sparkuser:sparkuser /opt/spark-3.3.0-bin-without-hadoop
ln -s /opt/spark-3.3.0-bin-without-hadoop /opt/spark
chown sparkuser:sparkuser /opt/spark

cd /opt/spark/conf
cp spark-env.sh.template spark-env.sh
```

vi spark-env.sh
export SPARK_LOCAL_IP=localhost
export HADOOP_HOME=/opt/hadoop
export HADOOP_CONF_DIR=/opt/hadoop/etc/hadoop
export YARN_CONF_DIR=/opt/hadoop/etc/hadoop
export SPARK_HOME=/opt/spark
export SPARK_CONF_DIR=/opt/spark/conf
export SPARK_LOG_DIR=/opt/spark/logs
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64

chmod +x conf/spark-env.sh
export SPARK_HOME=/opt/spark
cp spark-defaults.conf.template spark-defaults.conf
vi spark-defaults.conf
spark.master spark://localhost:7077
spark.eventLog.enabled true
spark.eventLog.dir hdfs://localhost:9000/user/sparkuser/spark/eventLog
spark.serializer org.apache.spark.serializer.KryoSerializer
spark.driver.memory 1g
spark.executor.extraJavaOptions -XX:+PrintGCDetails -Dkey=value -Dnumbers="one two three"

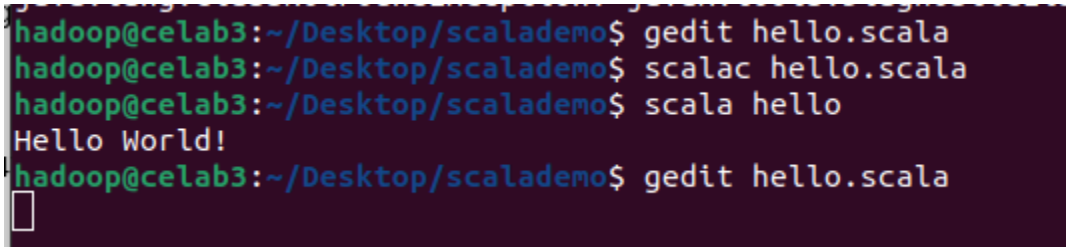
- **Creating Scala program**
    I. Using CLI
    II. USing IDE (eclipse,intelliJ)

1. Write the scala program in any editor and save with .scala extension

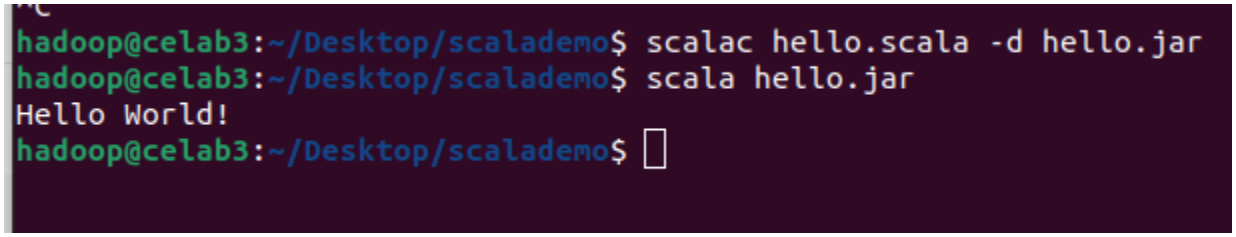
```
// Scala program to print Hello World!
object hello
{
   // Main Method
   def main(args: Array[String])
   {
   // prints Hello World
   println("Hello World!")
```

```
    }
  }
```

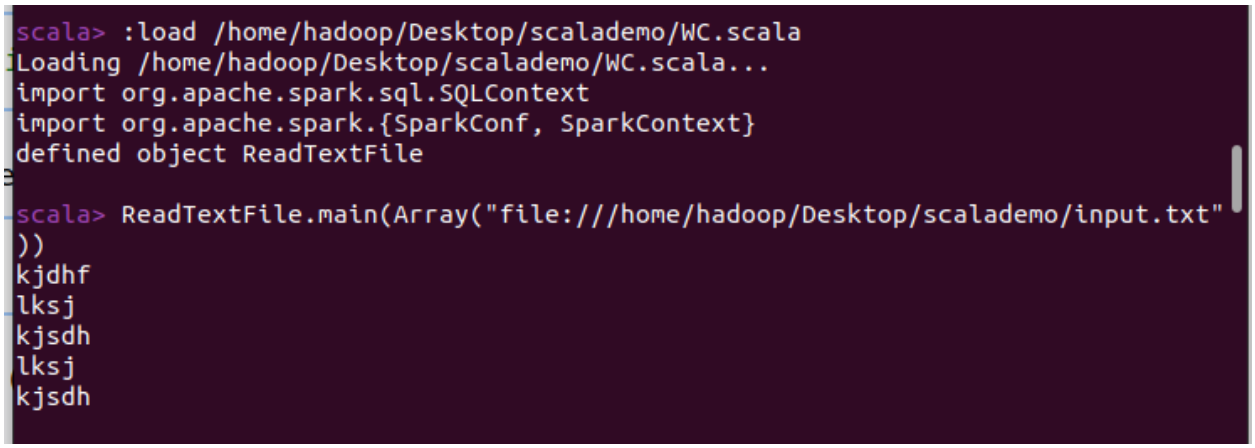
2. Compile the scala program using scalac
3. Run the scala program using scala

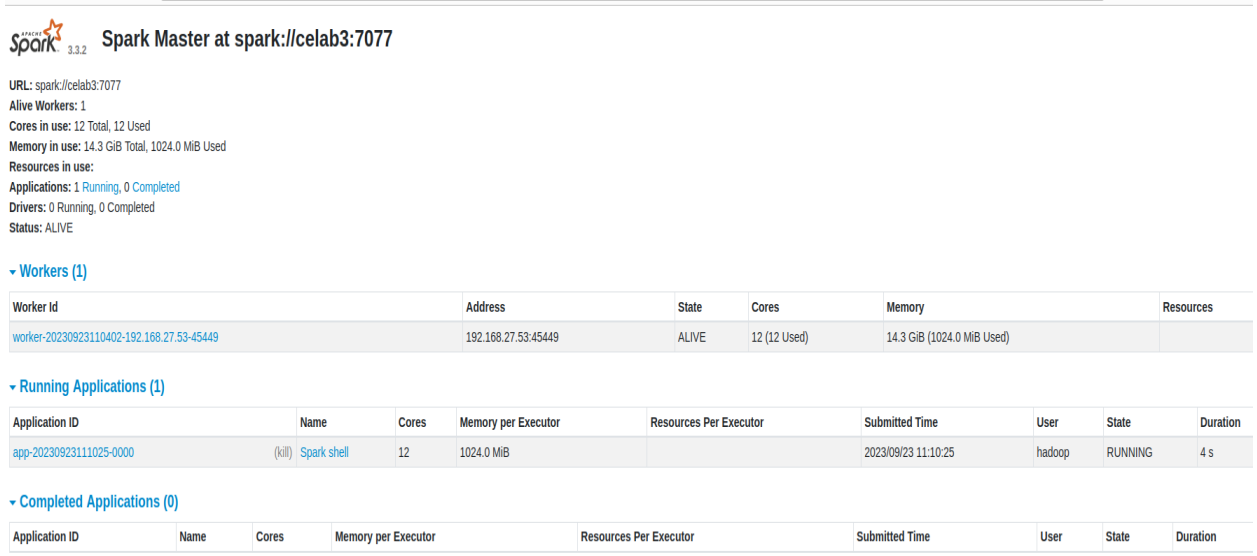


4. Creating jar files using CLI version



5. Load and run the scala file from shell





Spark Master at spark://celab3:7077

URL: spark://celab3:7077
Alive Workers: 1
Cores in use: 12 Total, 12 Used
Memory in use: 14.3 GiB Total, 1024.0 MiB Used
Resources in use:
Applications: 1 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (1)

Worker Id	Address	State	Cores	Memory	Resources
worker-20230923110402-192.168.27.53-45449	192.168.27.53:45449	ALIVE	12 (12 Used)	14.3 GiB (1024.0 MiB Used)	

Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20230923111025-0000 (kill)	Spark shell	12	1024.0 MiB		2023/09/23 11:10:25	hadoop	RUNNING	4 s

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Exercise:

1. Develop simple wordcount application with apache spark(using REPL and HDFS)
2. Develop Wordcount program using cluster mode(standalone or YARN mode)
3. Develop any custom map reduce problem using apache spark and hadoop.
4. Use IDE to develop application for standalone or cluster mode on cluster(Eclipse or any other IDE)-optional

References

1. Submitting application to spark https://spark.apache.org/docs/latest/submitting-applications.html
2. Spark configuration https://spark.apache.org/docs/latest/configuration.html
3. https://bigdataprogrammers.com/how-to-execute-scala-script-in-spark-submit-without-creating-jar/