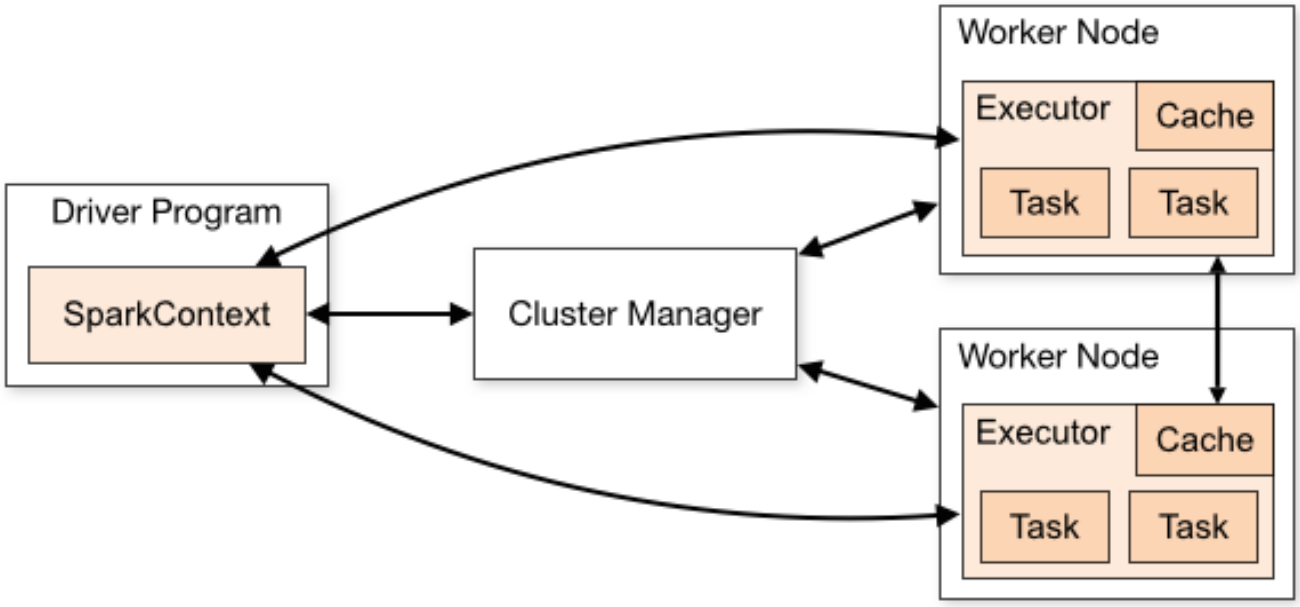
14 MAY 2019 / #SPARK

Deep-dive into Spark internals and architecture



by Jayvardhan Reddy

Apache Spark is an open-source distributed general-purpose cluster-computing framework. A spark application is a JVM process that's running a user code using the spark as a 3rd party library.

As part of this blog, I will be showing the way Spark works on Yarn architecture with an example and the various underlying background processes that are involved such as:

- Yarn Resource Manager, Application Master & launching of executors (containers).
- Setting up environment variables, job resources.
- CoarseGrainedExecutorBackend & Netty-based RPC.
- SparkListeners.
- Execution of a job (Logical plan, Physical plan).
- Spark-WebUI.

Spark Context

Spark context is the first level of entry point and the heart of any spark application. ***Spark-shell*** is nothing but a Scala-based REPL with spark binaries which will create an object sc called spark context.

We can launch the spark shell as shown below:

```
spark-shell --master yarn \ --conf spark.ui.port=12345 \ --num-exec
```

As part of the spark-shell, we have mentioned the num executors. They indicate the number of worker nodes to be used and the number of cores for each of these worker nodes to execute tasks in parallel.

Or you can launch spark shell using the default configuration.

```
spark-shell --master yarn
```

The configurations are present as part of **spark-env.sh**

```
cd /etc/spark/conf
vi spark-env.sh

# Options read in YARN client mode
#SPARK_EXECUTOR_INSTANCES="2" #Number of workers to start (Default: 2)
#SPARK_EXECUTOR_CORES="1" #Number of cores for the workers (Default: 1).
#SPARK_EXECUTOR_MEMORY="1G" #Memory per Worker (e.g. 1000M, 2G) (Default: 1G)
#SPARK_DRIVER_MEMORY="512M" #Memory for Master (e.g. 1000M, 2G) (Default: 512 Mb)
#SPARK_YARN_APP_NAME="spark" #The name of your application (Default: Spark)
```

Our Driver program is executed on the Gateway node which is nothing but a spark-shell. It will create a spark context and launch an application.

```
Spark context available as sc (master = yarn-client, app id = application_1540458187951_38909).
```

The spark context object can be accessed using **sc.**

```
scala> sc
res0: org.apache.spark.SparkContext = org.apache.spark.SparkContext@59449561
```

After the Spark context is created it waits for the resources. Once the resources are available, Spark context sets up internal services and establishes a connection to a Spark execution environment.

Yarn Resource Manager, Application Master & launching of executors (containers).

Once the Spark context is created it will check with the ***Cluster Manager*** and launch the ***Application Master*** i.e, launches a container


```
Container: container_e42_1540458187951_38909_01_000001 on wn01.jayReddy.com:45454
INFO ApplicationMaster: Registered signal handlers for [TERM, HUP, INT]
```

Once the Application Master is started it establishes a connection with the Driver.

```
INFO ApplicationMaster: Waiting for Spark driver to be reachable.
INFO ApplicationMaster: Driver now available: 172.16.1.113:44023
```

Next, the ApplicationMasterEndPoint triggers a proxy application to connect to the resource manager.

```
INFO ApplicationMaster$AMEndpoint: Add WebUI Filter. AddWebUIFilter(org.apache.hadoop.yarn.server.webproxy.amfilter.AmIpFilter,
Map(PROXY_HOSTS -> rm01.jayReddy.com, PROXY_URI_BASES -> http://rm01.jayReddy.com:19288/proxy/application_1540458187951_38909)
,/proxy/application_1540458187951_38909)
INFO RMProxy: Connecting to ResourceManager at rm01.jayReddy.com/172.16.1.106:8030
```

Now, the Yarn Container will perform the below operations as shown in the diagram.

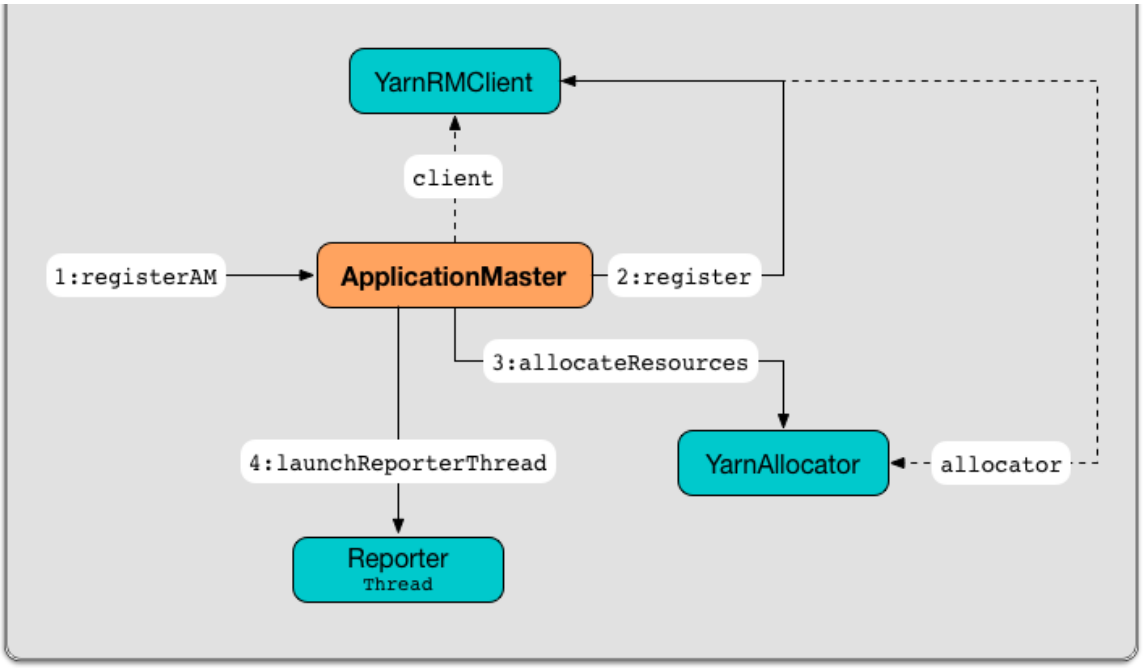


Image Credits: jaceklaskowski.gitbooks.io

ii) YarnRMClient will register with the Application Master.

```
INFO YarnRMClient: Registering the ApplicationMaster
```

iii) YarnAllocator: Will request 3 executor containers, each with 2 cores and 884 MB memory including 384 MB overhead

```
INFO YarnAllocator: Container request (host: Any, capability: <memory:884, vCores:2>)
```

iv) AM starts the Reporter Thread

```
INFO ApplicationMaster: Started progress reporter thread with (heartbeat : 5000, initial allocation : 200) intervals
```

Now the Yarn Allocator receives tokens from Driver to launch the Executor nodes and start the containers.

```
INFO AMRMClientImpl: Received new token for : wn03.jayReddy.com:45454
INFO YarnAllocator: Launching container container_e42_1540458187951_38909_01_000002 for on host wn03.jayReddy.com
INFO YarnAllocator: Launching ExecutorRunnable. driverUrl:
spark://CoarseGrainedScheduler@172.16.1.113:44023,  executorHostname: wn03.jayReddy.com
INFO ExecutorRunnable: Starting Executor Container
INFO YarnAllocator: Received 3 containers from YARN, launching executors on 3 of them.
```

Setting up environment variables, job resources & launching containers.

Every time a container is launched it does the following 3 things in each of these.

- Setting up env variables

Spark Runtime Environment (SparkEnv) is the runtime environment with Spark's services that are used to interact with each other in order to establish a distributed computing platform for a Spark application.

```
import org.apache.spark._
scala> SparkEnv.get
res0: org.apache.spark.SparkEnv = org.apache.spark.SparkEnv@49322d04
```

```
echo "Setting up env variables"
export SPARK_YARN_STAGING_DIR=".sparkStaging/application_1540458187951_38909"
export NM_HOST="wn01.jayReddy.com"
export USER="jvanchir"
export SPARK_YARN_CACHE_FILES_FILE_SIZES="194710318"
export JAVA_HOME=${JAVA_HOME:-"/usr/jdk64/jdk1.8.0_77"}
export LOGNAME="jvanchir"
export HADOOP_CONF_DIR=${HADOOP_CONF_DIR:-"/usr/hdp/2.6.5.0-292/hadoop/conf"}
```

- Setting up job resources

```
echo "Setting up job resources"
ln -sf "/hdp02/hadoop/yarn/local/filecache/2702/spark-hdp-assembly.jar" "__spark__.jar"
ln -sf "/hdp01/hadoop/yarn/local/usercache/jvanchir/filecache/4020/__spark_conf__1756319926205361137.zip" "__spark_conf__"
```

- Launching container

```
echo "Launching container"
exec /bin/bash -c $JAVA_HOME/bin/java -server -Xmx512m -Djava.io.tmpdir=$PWD/tmp -Dhdp.version=2.6.5.0-292
-Dspark.yarn.app.container.log.dir=/hdp01/hadoop/yarn/log/application_1540458187951_38909/container_e42_1540458187951_38909_01_000001
org.apache.spark.deploy.yarn.ExecutorLauncher --arg '172.16.1.113:44023' --executor-memory 500m --executor-cores 2
--properties-file $PWD/__spark_conf__/__spark_conf__.properties 1> /hdp01/hadoop/yarn/log/application_1540458187951_38909/container_e
/hdp01/hadoop/yarn/log/application_1540458187951_38909/container_e42_1540458187951_38909_01_000001/stderr"
```

YARN executor launch context assigns each executor with an executor id to identify the corresponding executor (via Spark WebUI) and starts a CoarseGrainedExecutorBackend.

```
-Dspark.yarn.app.container.log.dir=<LOG_DIR> org.apache.spark.executor.CoarseGrainedExecutorBackend
--driver-url spark://CoarseGrainedScheduler@172.16.1.113:44023 --executor-id 1 --hostname wn03.jayReddy.com
--cores 2 --app-id application_1540458187951_38909
```

CoarseGrainedExecutorBackend & Netty-based RPC.

After obtaining resources from Resource Manager, we will see the executor starting up

```
INFO Remoting: Starting remoting
INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkExecutorActorSystem@wn03.jayReddy.com:60073]
INFO Utils: Successfully started service 'sparkExecutorActorSystem' on port 60073.
```

CoarseGrainedExecutorBackend is an ExecutorBackend that controls

driver.

When ExecutorRunnable is started, CoarseGrainedExecutorBackend registers the Executor RPC endpoint and signal handlers to communicate with the driver (i.e. with CoarseGrainedScheduler RPC endpoint) and to inform that it is ready to launch tasks.

```
INFO CoarseGrainedExecutorBackend: Registered signal handlers for [TERM, HUP, INT]
INFO CoarseGrainedExecutorBackend: Connecting to driver: spark://CoarseGrainedScheduler@172.16.1.113:44023
INFO CoarseGrainedExecutorBackend: Successfully registered with driver
INFO Executor: Starting executor ID 1 on host wn03.jayReddy.com
```

Netty-based RPC - It is used to communicate between worker nodes, spark context, executors.

```
INFO Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlockTransferService' on port 38966.
INFO NettyBlockTransferService: Server created on 38966
```

NettyRPCEndPoint is used to track the result status of the worker node.

RpcEndpointAddress is the logical address for an endpoint registered to an RPC Environment, with RpcAddress and name.

It is in the format as shown below:

```
spark://[name]@[rpcAddress.host]:[rpcAddress.port].
```

This is the first moment when CoarseGrainedExecutorBackend

RpcEnv.

```
INFO MapOutputTrackerWorker: Doing the fetch; tracker endpoint = NettyRpcEndpointRef(spark://MapOutputTracker@172.16.1.113:44023)
```

SparkListeners

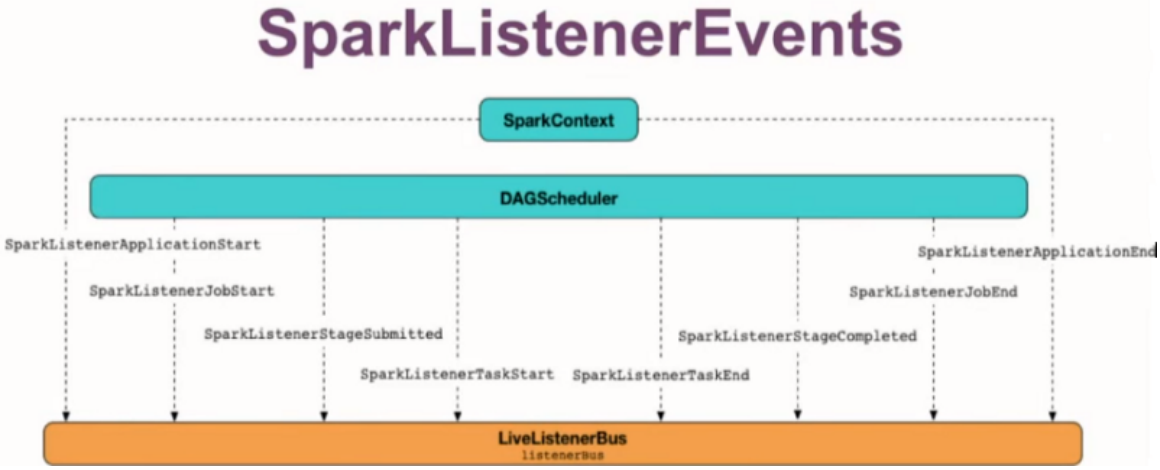


Image Credits: jaceklaskowski.gitbooks.io

SparkListener (Scheduler listener) is a class that listens to execution events from Spark's DAGScheduler and logs all the event information of an application such as the executor, driver allocation details along with jobs, stages, and tasks and other environment properties changes.

SparkContext starts the LiveListenerBus that resides inside the driver. It registers JobProgressListener with LiveListenerBus which collects all the data to show the statistics in spark UI.

By default, only the listener for WebUI would be enabled but if we want to add any other listeners then we can use **spark.extraListeners.**

i) StatsReportListener

ii) EventLoggingListener

EventLoggingListener: If you want to analyze further the performance of your applications beyond what is available as part of the Spark history server then you can process the event log data. Spark Event Log records info on processed jobs/stages/tasks. It can be enabled as shown below...

```
spark.eventLog.enabled true
spark.eventLog.dir hdfs://namenode/shared/spark-logs
    (OR)
spark.eventLog.enabled=true
spark.eventLog.dir=hdfs://user/spark/applicationHistory
```

The event log file can be read as shown below

- The Spark driver logs into job workload/perf metrics in the spark.evenLog.dir directory as JSON files.
- There is one file per application, the file names contain the application id (therefore including a timestamp) application_1540458187951_38909.

```
val df = spark.read.json("/user/spark/applicationHistory/application_1540458187951_38909")
df.printSchema

scala> df.select("Event").groupBy("Event").count.show(20,false)
```

It shows the type of events and the number of entries for each.

```
+------------------------------------------------------------------+-----+
|Event                                                             |count|
+------------------------------------------------------------------+-----+
|org.apache.spark.sql.execution.ui.SparkListenerSQLExecutionEnd    |3    |
|SparkListenerTaskStart                                            |249  |
|SparkListenerBlockManagerAdded                                    |4    |
|SparkListenerJobStart                                             |2    |
|SparkListenerStageCompleted                                       |5    |
|SparkListenerJobEnd                                               |2    |
|SparkListenerLogStart                                             |1    |
|SparkListenerExecutorAdded                                        |4    |
|org.apache.spark.sql.execution.ui.SparkListenerSQLExecutionStart  |3    |
|SparkListenerEnvironmentUpdate                                    |1    |
|SparkListenerStageSubmitted                                       |5    |
|SparkListenerTaskEnd                                              |249  |
|SparkListenerApplicationStart                                     |1    |
+------------------------------------------------------------------+-----+
```

Now, let's add ***StatsReportListener*** to the spark.extraListeners and check the status of the job.

Enable INFO logging level for org.apache.spark.scheduler.StatsReportListener logger to see Spark events.

```
Add the following line to conf/log4j.properties:
log4j.logger.org.apache.spark.scheduler.StatsReportListener=INFO
```

To enable the listener, you register it to SparkContext. It can be done in two ways.

i) Using SparkContext.addSparkListener(listener: SparkListener) method inside your Spark application.

Click on the link to implement custom listeners - **CustomListener**

ii) Using the conf command-line option

```
$ ./bin/spark-shell --conf  spark.extraListeners=org.apache.spark.scheduler.StatsReportListener
INFO SparkContext: Registered listener org.apache.spark.scheduler.StatsReportListener
```

Let's read a sample file and perform a count operation to see the StatsReportListener.

```
scala> spark.read.text("sampletxt").count
...
INFO StatsReportListener: Finished stage: Stage(0, 0); Name: 'count at <console>:24'; Status: succeeded; numTasks: 1; Took: 212 msec
INFO StatsReportListener: task runtime:(count: 1, mean: 198.000000, stdev: 0.000000, max: 198.000000, min: 198.000000)
INFO StatsReportListener:   0%  5%  10% 25% 50% 75% 90% 95% 100%
INFO StatsReportListener:   198.0 ms  198.0 ms  198.0 ms  198.0 ms  198.0 ms  198.0 ms  198.0 ms  198.0 ms  198.0 ms
INFO StatsReportListener: shuffle bytes written:(count: 1, mean: 59.000000, stdev: 0.000000, max: 59.000000, min: 59.000000)
INFO StatsReportListener:   0%  5%  10% 25% 50% 75% 90% 95% 100%
```

Execution of a job (Logical plan, Physical plan).

In Spark, RDD (*resilient distributed dataset*) is the first level of the abstraction layer. It is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs can be created in 2 ways.

i) P*arallelizing* an existing collection in your driver program

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

ii) Referencing a dataset in an external storage system

RDDs are created either by using a file in the Hadoop file system, or an existing Scala collection in the driver program, and transforming it.

Let's take a sample snippet as shown below

```
val tokenized = sc.textFile("/user/jvanchir/sampletxt.txt").flatMap(_.split(" "))
val wordCounts = tokenized.map((_, 1)).reduceByKey(_ + _)
wordCounts.collect
```

The execution of the above snippet takes place in 2 phases.

6.1 Logical Plan: In this phase, an RDD is created using a set of transformations, It keeps track of those transformations in the driver program by building a computing chain (a series of RDD)as a Graph of transformations to produce one RDD called a ***Lineage Graph***.

Transformations can further be divided into 2 types

- **Narrow transformation:** A pipeline of operations that can be executed as one stage and does not require the data to be shuffled across the partitions — for example, Map, filter, etc..

```
scala> val tokenized = sc.textFile("/user/jvanchir/sampletxt.txt").flatMap(_.split(" "))
```

Now the data will be read into the driver using the broadcast variable.

```
19/01/08 12:25:46 INFO SparkContext: Created broadcast 0 from textFile at <console>:27
tokenized: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at flatMap at <console>:27
```

- **Wide transformation:** Here each operation requires the data to be shuffled, henceforth for each wide transformation a new stage will be created — for example, reduceByKey, etc..

```
scala> val wordCounts = tokenized.map((_, 1)).reduceByKey(_ + _)
19/01/08 12:26:14 INFO FileInputFormat: Total input paths to process : 1
wordCounts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:29
```

We can view the lineage graph by using ***toDebugString***

```
scala> wordCounts.toDebugString
res0: String =
(2) ShuffledRDD[4] at reduceByKey at <console>:29 []
 +-(2) MapPartitionsRDD[3] at map at <console>:29 []
    |  MapPartitionsRDD[2] at flatMap at <console>:27 []
    |  /user/jvanchir/sampletxt.txt MapPartitionsRDD[1] at textFile at <console>:27 []
    |  /user/jvanchir/sampletxt.txt HadoopRDD[0] at textFile at <console>:27 []
```

6.2 Physical Plan: In this phase, once we trigger an action on the RDD, The ***DAG Scheduler*** looks at RDD lineage and comes up with the best execution plan with stages and tasks together with TaskSchedulerImpl and execute the job into a set of tasks parallelly.

```
  wordCounts.collect
```

Once we perform an action operation, the SparkContext triggers a job and registers the RDD until the first stage (i.e, before any wide transformations) as part of the DAGScheduler.

```
INFO DAGScheduler: Registering RDD 3 (map at <console>:29)
INFO DAGScheduler: Got job 0 (collect at <console>:32) with 2 output partitions
INFO DAGScheduler: Final stage: ResultStage 1 (collect at <console>:32)
INFO DAGScheduler: Parents of final stage: List(ShuffleMapStage 0)
```

Now before moving onto the next stage (Wide transformations), it will check if there are any partition data that is to be shuffled and if it has any missing parent operation results on which it depends, if any such stage is missing then it re-executes that part of the operation by making use of the DAG(Directed Acyclic Graph) which makes it Fault tolerant.

```
INFO DAGScheduler: Missing parents: List(ShuffleMapStage 0)
INFO DAGScheduler: Submitting ShuffleMapStage 0 (MapPartitionsRDD[3] at map at <console>:29), which has no missing parents
INFO MemoryStore: Block broadcast_1 stored as values in memory (estimated size 4.3 KB, free 510.7 MB)
INFO MemoryStore: Block broadcast_1_piece0 stored as bytes in memory (estimated size 2.4 KB, free 510.7 MB)
INFO BlockManagerInfo: Added broadcast_1_piece0 in memory on 172.16.1.113:35918 (size: 2.4 KB, free: 511.1 MB)
```

In the case of missing tasks, it assigns tasks to executors.

```
INFO DAGScheduler: Submitting 2 missing tasks from ShuffleMapStage 0 (MapPartitionsRDD[3] at map at <console>:29)
INFO YarnScheduler: Adding task set 0.0 with 2 tasks
INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0, wn03.jayReddy.com, partition 0,NODE_LOCAL, 2149 bytes)
INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1, wn03.jayReddy.com, partition 1,NODE_LOCAL, 2149 bytes)
```

Each task is assigned to CoarseGrainedExecutorBackend of the executor.

```
INFO CoarseGrainedExecutorBackend: Got assigned task 0
INFO CoarseGrainedExecutorBackend: Got assigned task 1
INFO Executor: Running task 0.0 in stage 0.0 (TID 0)
INFO Executor: Running task 1.0 in stage 0.0 (TID 1)
```

It gets the block info from the Namenode.

```
INFO HadoopRDD: Input split: hdfs://nn01.jayReddy.com:8020/user/jvanchir/sampletxt.txt:14+14
INFO HadoopRDD: Input split: hdfs://nn01.jayReddy.com:8020/user/jvanchir/sampletxt.txt:0+14
```

now, it performs the computation and returns the result.

```
INFO BlockManagerInfo: Added broadcast_1_piece0 in memory on wn03.jayReddy.com:38966 (size: 2.4 KB, free: 134.6 MB)
INFO BlockManagerInfo: Added broadcast_0_piece0 in memory on wn03.jayReddy.com:38966 (size: 30.7 KB, free: 134.6 MB)
INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 1269 ms on wn03.jayReddy.com (1/2)
INFO TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1) in 1250 ms on wn03.jayReddy.com (2/2)
INFO YarnScheduler: Removed TaskSet 0.0, whose tasks have all completed, from pool
INFO DAGScheduler: ShuffleMapStage 0 (map at <console>:29) finished in 1.282 s
```

Next, the DAGScheduler looks for the newly runnable stages and triggers the next stage (reduceByKey) operation.

```
INFO DAGScheduler: looking for newly runnable stages
INFO DAGScheduler: Submitting ResultStage 1 (ShuffledRDD[4] at reduceByKey at <console>:29), which has no missing parents
```

The ShuffleBlockFetcherIterator gets the blocks to be shuffled.

```
INFO ShuffleBlockFetcherIterator: Getting 0 non-empty blocks out of 2 blocks
INFO ShuffleBlockFetcherIterator: Getting 1 non-empty blocks out of 2 blocks
INFO ShuffleBlockFetcherIterator: Started 0 remote fetches in 3 ms
INFO ShuffleBlockFetcherIterator: Started 0 remote fetches in 3 ms
```

Now the reduce operation is divided into 2 tasks and executed.

```
INFO MapOutputTrackerMaster: Size of output statuses for shuffle 0 is 158 bytes
INFO TaskSetManager: Finished task 0.0 in stage 1.0 (TID 3) in 54 ms on wn03.jayReddy.com (1/2)
INFO TaskSetManager: Finished task 1.0 in stage 1.0 (TID 2) in 58 ms on wn03.jayReddy.com (2/2)
INFO DAGScheduler: ResultStage 1 (collect at <console>:32) finished in 0.058 s
```

On completion of each task, the executor returns the result back to the driver.

```
INFO Executor: Finished task 0.0 in stage 1.0 (TID 3). 1170 bytes result sent to driver
INFO Executor: Finished task 1.0 in stage 1.0 (TID 2). 1342 bytes result sent to driver
```

Once the Job is finished the result is displayed.

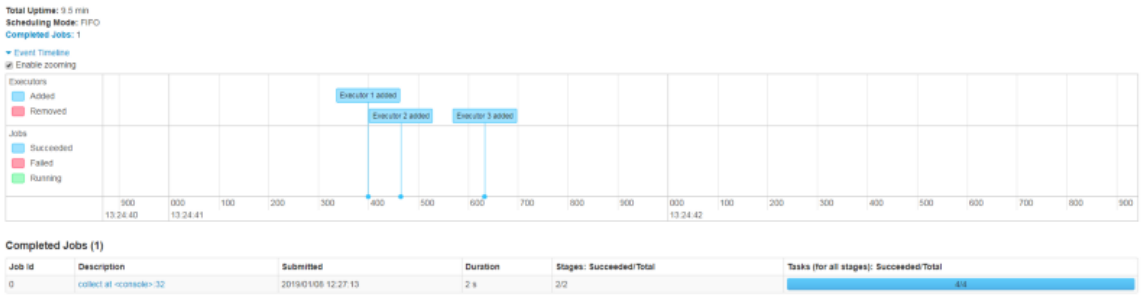
```
  INFO DAGScheduler: Job 0 finished: collect at <console>:32, took 1.588329 s
res1: Array[(String, Int)] = Array((Jack,Jone,Mars,Hilton,Maan,,1))
```

Spark-WebUI

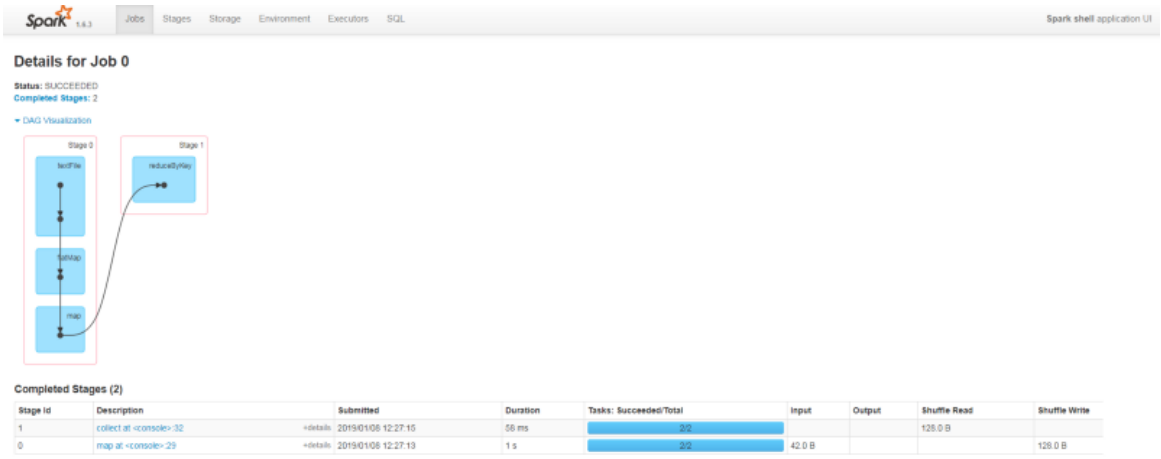
Spark-UI helps in understanding the code execution flow and the time taken to complete a particular job. The visualization helps in finding out any underlying problems that take place during the execution and optimizing the spark application further.

We will see the Spark-UI visualization as part of the previous **step 6.**

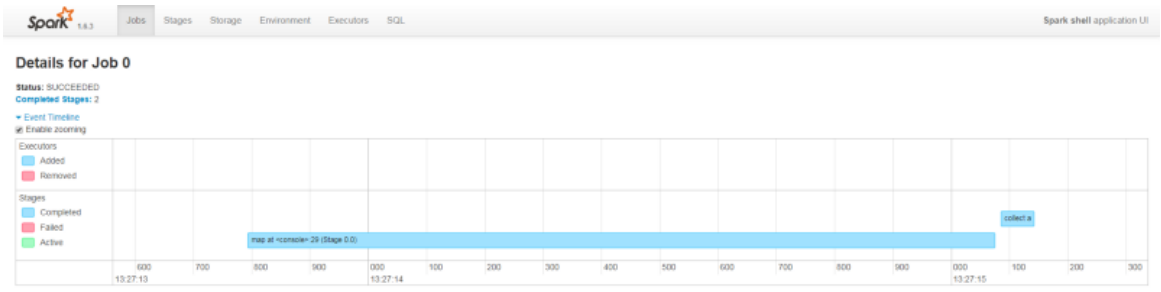
Once the job is completed you can see the job details such as the number of stages, the number of tasks that were scheduled during the job execution of a Job.



On clicking the completed jobs we can view the DAG visualization i.e, the different wide and narrow transformations as part of it.

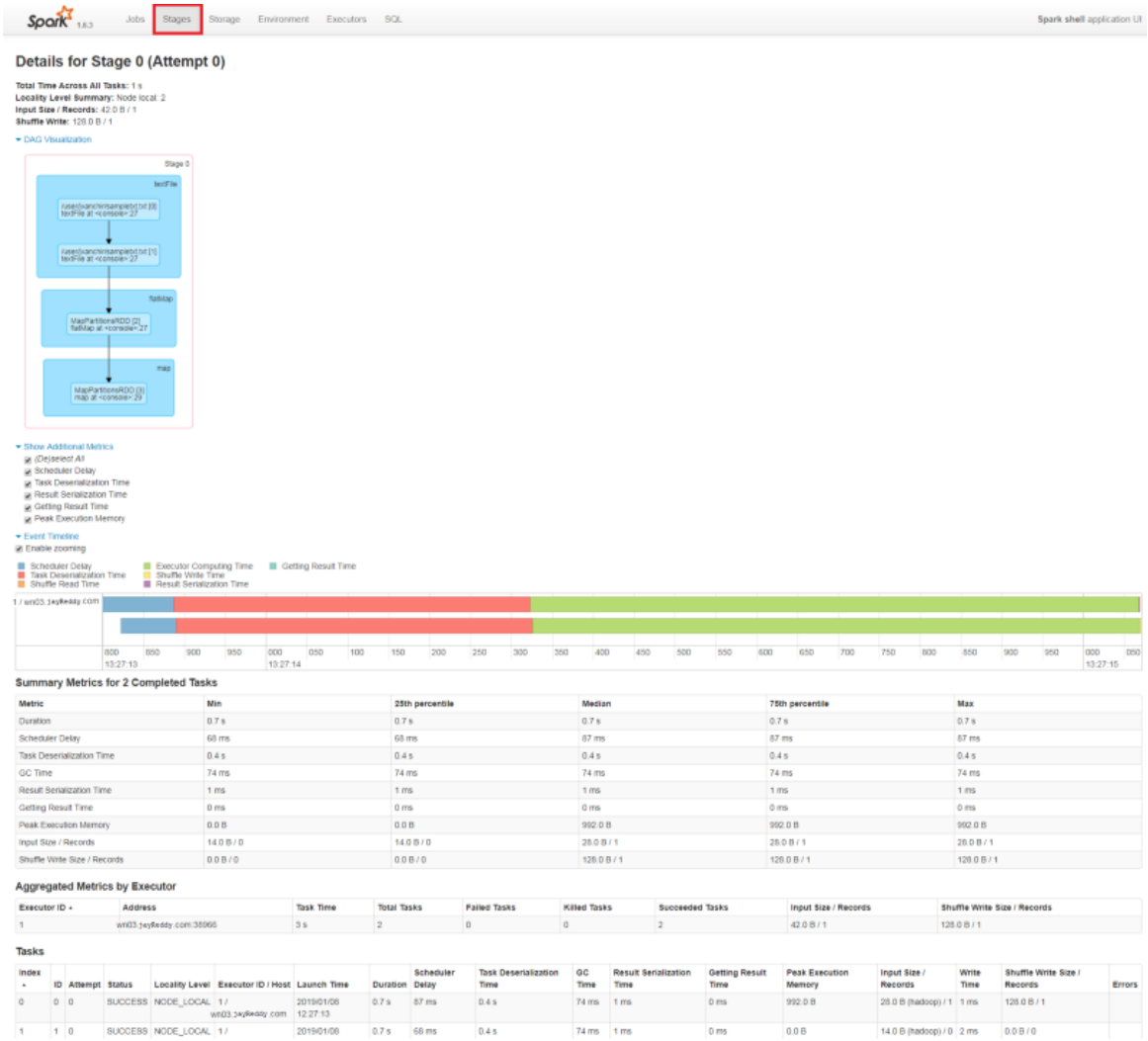


You can see the execution time taken by each stage.



On clicking on a Particular stage as part of the job, it will show the complete details as to where the data blocks are residing, data size,

particular task. It also shows the number of shuffles that take place.



Further, we can click on the Executors tab to view the Executor and driver used.

Now that we have seen how Spark works internally, you can determine the flow of execution by making use of Spark UI, logs and

the submission of a Spark job.
Note: The commands that were executed related to this post are added as part of my GIT account.

Similarly, you can also read more here:

- Sqoop Architecture in Depth with **code.**
- HDFS Architecture in Depth with **code**.
- Hive Architecture in Depth with **code**.

If you would like too, you can connect with me on LinkedIn — Jayvardhan Reddy.

If you enjoyed reading it, you can click the clap and let others know about it. If you would like me to add anything else, please feel free to leave a response ?

PHP Example