Ref

GUIDE TO APACHE SPARK EXECUTION

Unraveling the Staged Execution in Apache Spark

Stage in Spark represents a logical unit of parallel computation. Many such stages assembled together builds the execution skeleton of a Spark application. This story tries to unravel the concept of Spark stage and describes important related aspects.

Ajay Gupta May 31, 2020 · 8 min read ★

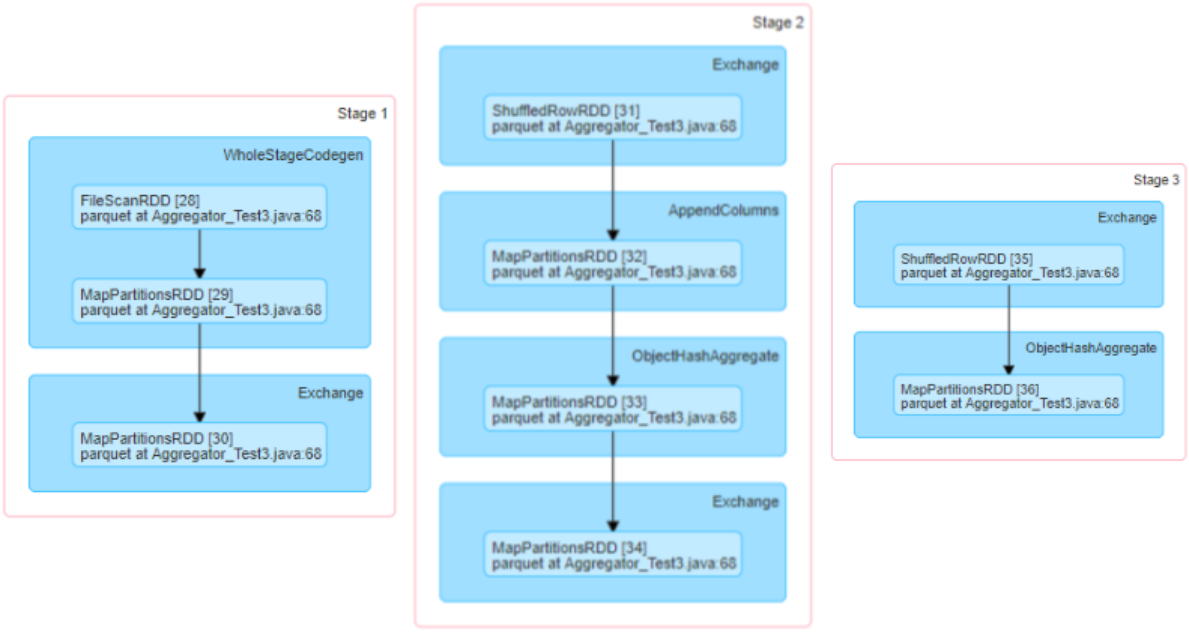
A Spark stage can be understood as a compute block to compute data partitions of a distributed collection, the compute block being able to execute in parallel in a cluster of computing nodes. Spark builds parallel execution flow for a Spark application using single or multiple stages. Stages provides modularity, reliability and resiliency to spark application execution. Below are the various important aspects related to Spark Stages:

Stages are created, executed and monitored by DAG scheduler: Every running Spark application has a DAG scheduler instance associated with it. This scheduler create stages in response to submission of a Job, where a Job essentially represents a RDD execution plan (also called as RDD DAG) corresponding to a action taken in a Spark application. Multiple Jobs could be submitted to DAG scheduler if multiple actions are taken in a Spark application. For each of Job submitted to it, DAG scheduler creates one or more stages, builds a stage DAG to list out the stage dependency graph, and then plan execution schedule for the created stages in accordance with stage DAG. Also, the scheduler monitors the status of the stage execution completion which could turn out to be success, partial-success, or failure. Accordingly, the scheduler attempt for stage re-execution, conclude Job failure/success, or schedule dependent stages as per stage DAG.

Below is a sample RDD execution plan (DAG) against a Job:

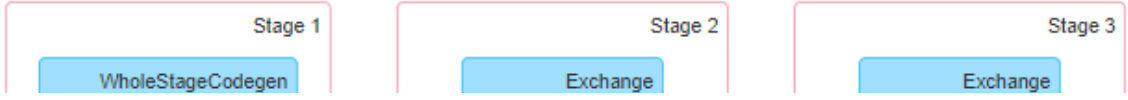
```
(200) MapPartitionsRDD[36]
 | ShuffledRowRDD[35]
 +-(500) MapPartitionsRDD[34]
 | MapPartitionsRDD[33]
 | MapPartitionsRDD[32]
 | ShuffledRowRDD[31]
 +-(3) MapPartitionsRDD[30]
 | MapPartitionsRDD[29]
 | FileScanRDD[28]
```

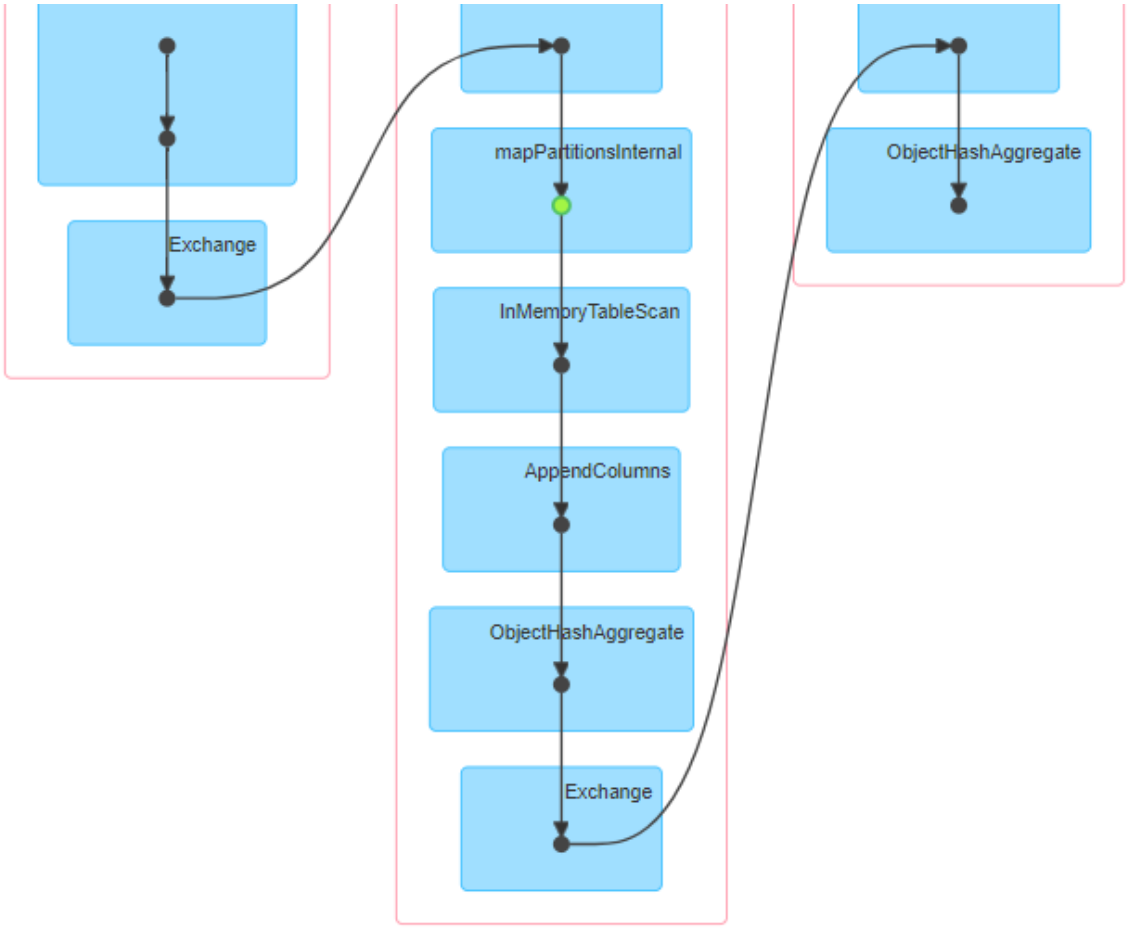
Below three stages are created by DAG scheduler for the above RDD execution plan:



Fig(2): Three stages created against the RDD execution plan shown in Fig(2)

Below is stage DAG created by the DAG scheduler against the above stages which clearly indicates the inter stage dependencies:





Fig(3): Stage DAG created for the three stages shown in Fig(2) as per the RDD execution plan of Job shown in Fig(1)

> *Every stage created by the DAG scheduler bears a unique ID across Jobs in a Spark application. Further, a stage is scheduled for execution in accordance with stage DAG, meaning a stage is scheduled for execution after all the dependent stages (as listed in stage DAG) are already computed. Two stages can be executed simultaneously if they are not inter dependent on each other and their all other dependent stages are already computed.*

Stages are created on shuffle boundaries: DAG scheduler creates multiple stages by splitting a RDD execution plan/DAG (associated with a Job) at shuffle boundaries indicated by

ShuffleRDD's in the plan. In this splitting process, therefore, a segment of RDD execution plan,essentially a RDD pipeline, becomes the part of a stage. Shuffle is necessitated for wide transformations mentioned in a Spark application, examples of which includes aggregation, join, or repartition operations.

Below is a illustration of stage creation at various shuffle boundaries.

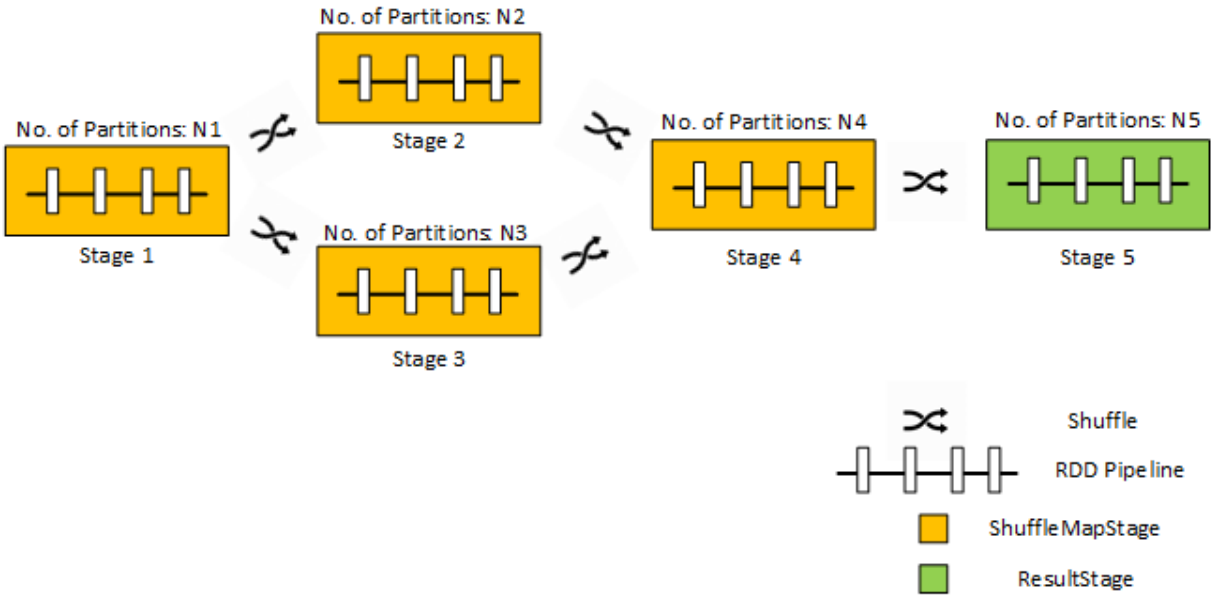


Illustration of stage creation at shuffle boundaries.

Also, in the example shown in Fig (1), there are two shuffles happening as indicated by two ShuffleRowRDD's in the RDD execution plan, and therefore three stages are created as shown in Fig (2). As evident from Fig (2), each of three stage contains a RDD pipeline (segment of Job's original RDD execution plan/DAG).

> *In case there is no shuffle required in a submitted Job, DAG scheduler will only create and schedule a single stage for the Job*

Stages are of either type, ShuffleMapStage or ResultStage: Stages of type ShuffleMapStage are intermediate stages of a Job execution plan, the Job being triggered against an action mentioned in a Spark application. A ShuffleMapStage essentially produces shuffle data files which are consumed by ShuffledRDD in the subsequent stage(s). But, before producing the output data, a ShuffleMapStage has to execute the segment of Job's RDD execution plan (essentially a RDD pipeline) contained in the ShuffleMapStage. The amount of shuffle data produced by a ShuffleMapStage is available as a stage metric called as ShuffleWrite. Also, since SortShuffle process is mostly used to produce shuffle data files, the number of shuffle data files produced by a ShuffleMapStage is equal to number of data partitions computed by the RDD pipeline of the stage.

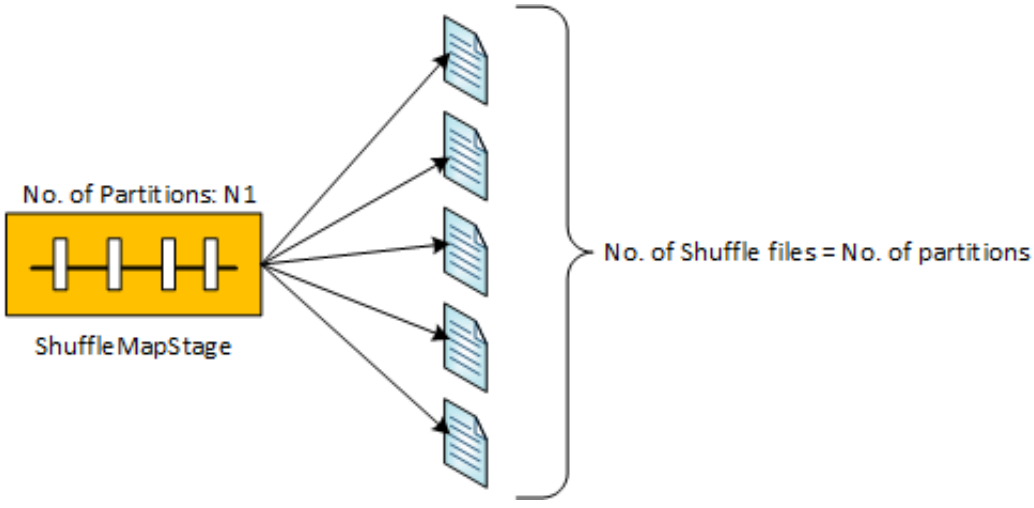


Illustration of ShuffleMapStage producing a set of shuffle files. The number of shuffle files are equal to the number of partitions in the ShuffleMapStage

A ResultStage is a final stage in a Job execution plan, in which a function (corresponding to the action initiating the Job) is applied to all or some partitions, the partitions being computed by executing the segment of RDD execution plan contained in the ResultStage. This function produces the final desired output expected from the execution of corresponding action in a spark application. A list of possible actions for a spark application are listed here.

> *A Job, triggered by an action in a Spark application, either consists of a single ResultStage only, or a combination of intermediate ShuffleMapStage(s) along with a single ResultStage. However, for adaptive query planning or adaptive scheduling, some special Jobs consisting of only ShuffleMapStage(s) could be executed by DAG scheduler on request(s).*
>
> *Data to a ShuffleMapStage or ResultStage is fed individually or in combination from input files, shuffle files from previous ShuffleMap stages, or cached RDDs.*

Each Stage has a associated Taskset(s) for execution: For an execution attempt of a stage, DAG scheduler creates a corresponding Taskset. A stage Taskset is basically a collection of tasks, where each task executes the stage RDD pipeline for a specific data partition and produces the desired output.

Similar to stage types, a task in a stage Taskset is of either type, ShuffleMapTask or ResultTask. ShuffleMapTasks are created for

ShuffleMapStage while ResultTasks are created for ResultStage.

Taskset, created for an execution attempt of stage, is submitted to Task scheduler instance of the Spark application. Task scheduler, in turn, schedules the execution of each task (contained in the Taskset) at an appropriate location in the cluster. After all the tasks in the Taskset are executed, the corresponding execution status of each task in the Taskset is reported back to the DAG scheduler. Accordingly, the DAG scheduler marks the stage execution as complete success, partial success or complete failure.

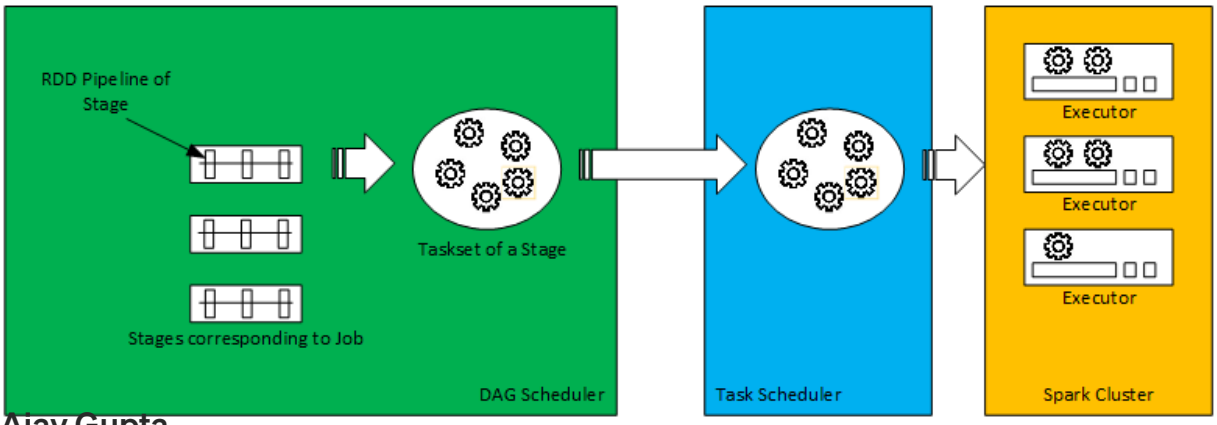


Illustration of creation of Taskset for a Stage by DAG scheduler, submission of Taskset to Task scheduler and scheduling of Tasks (of submitted Taskset) for execution on executors of in Spark cluster by DAG scheduler

In case of partial success, stage execution is re-attempted with a partial Taskset consisting of only those tasks that were failed earlier. In case of complete failure, stage execution is re-attempted with a full-fledged Taskset.

> *A Stage is re-attempted only for a certain number of times, and when all the re-attempts together could not mark the stage execution as complete success, the stage execution is marked as failed leading to execution failure of the corresponding Job submitted to the DAG scheduler.*
>
> *Also, most of the times, a stage gets failed partially due to unavailability of some or all of the shuffle data files produced by the parent stage(s). This leads to partial re-execution of parent stage(s) (in order to compute the missing shuffle files) before the stage reported for failure is reattempted again. Further, this re-execution could also reach to stages present at deeper levels into the parental stage ancestry if there are missed shuffle files at successive levels in the ancestry. Shuffle files become unavailable when the executors hosting the files may get lost due to memory overruns or forced killing by cluster manager. Also, the shuffle files remain unavailable when the corresponding ShuffledRDD gets garbage collected.*

Stages computation can be skipped at times: DAG scheduler could decide to skip the computation of a ShuffleMap stage in a submitted Job if a similar stage is already being computed against a previous job submitted to the scheduler. The two stages are deemed to be similar if they are executing the same RDD pipeline. This skipping is possible because the shuffle output files are retained in the disk until the shuffled RDD reference is retained in the Spark application.

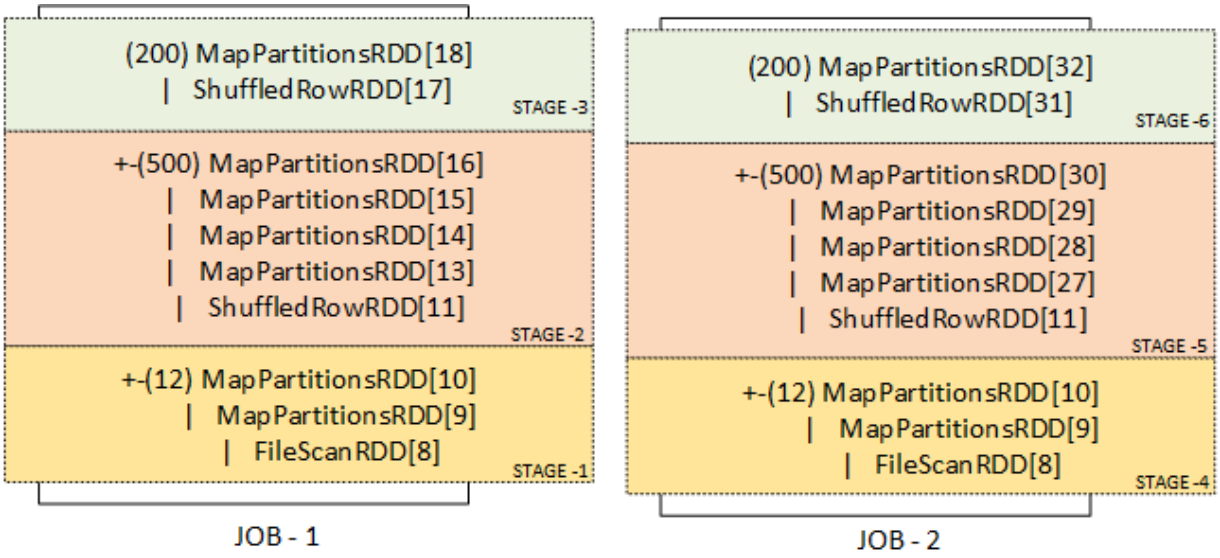
Also, in another instance, DAG scheduler could decide on skipping the computation of a ShuffleMap stage if the Shuffled RDD required by the dependent downstream stage(s) is already cached, the caching being done during the execution of another stage against a previous job submitted to the scheduler.

Below is a sample Spark application for illustration of stage skipping by DAG scheduler, this particular application caches a re-partitioned dataset (on line 10) which is being used in two File write actions, on line 15 and 20 respectively :

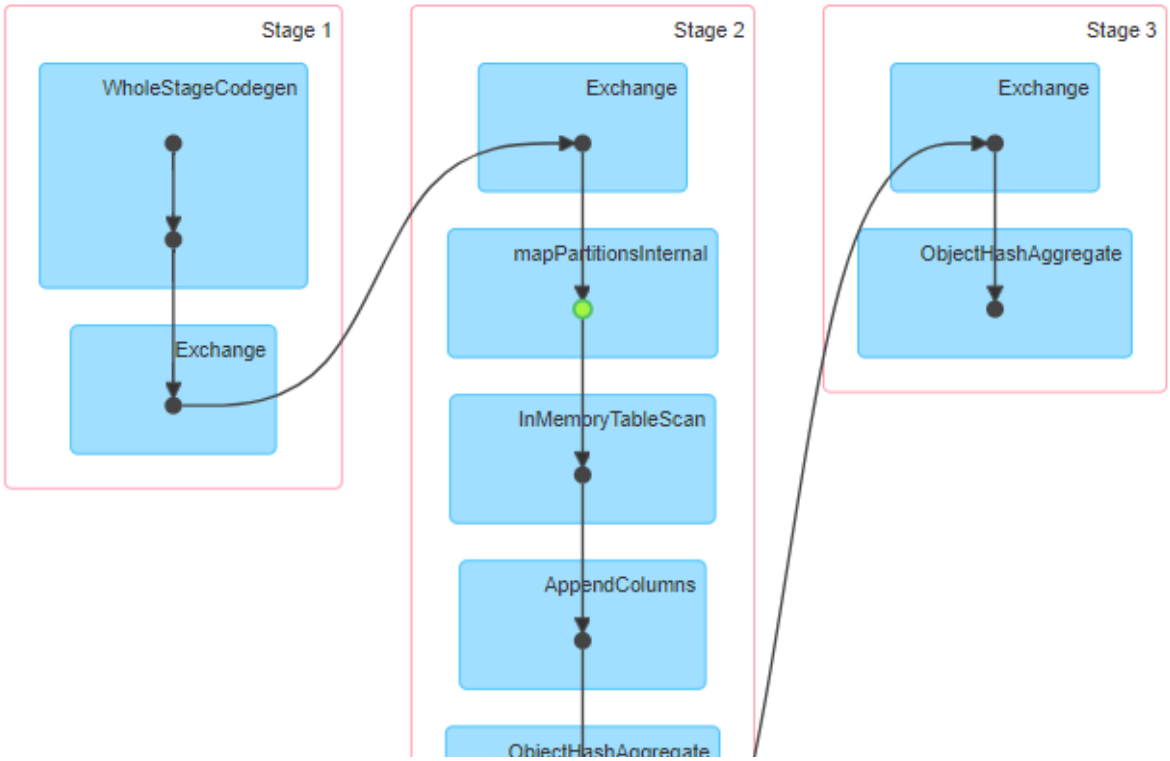
```
1  public void runJob(Configuration configuration) throws Exception {
2    logger.info("Inside @run_22 ");
3    sparkSession = SparkUtils.getSparkClient().getSparkSession("Stage_Test"
4    String path = "/user/stage/input/*.csv";
5    Dataset<Employee> ds_load = sparkSession.read().format("csv").option("h
6                              .select(functions.col("empId").cast(DataTypes.
7                               functions.col("deptNo").cast(DataTypes.LongTy
8                              .as(Encoders.bean(Employee.class));
9    Dataset<Employee> ds = ds_load.repartition(500); /* Cause Shuffle */
10   ds.cache();
11
12   Dataset<Tuple2<Long, Employee>> agg = ds.groupByKey((MapFunction<Employ
13                                            Encoders.LONG()).agg(new MyAv
14
15   agg.write().mode(SaveMode.Overwrite).parquet("/user/stage/job1"); /* Ac
16
17   Dataset<Tuple2<Long, Employee>> agg1 = ds.groupByKey((MapFunction<Emplo
18                                            Encoders.LONG()).agg(new MyAv
19
20  agg1.write().mode(SaveMode.Overwrite).parquet("/user/stage/job1"); /* Ac
```

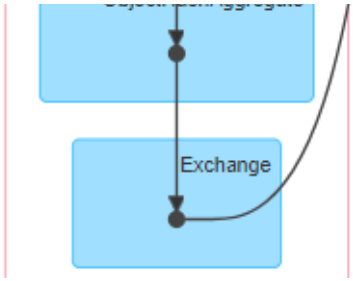
Below are the RDD execution plans (with marking of stages) belonging to two different Jobs (triggered by two actions) in

the above Spark application. In the application, dataset 'ds' is cached causing the corresponding 'ShuffledRowRDD[11]' to be cached.

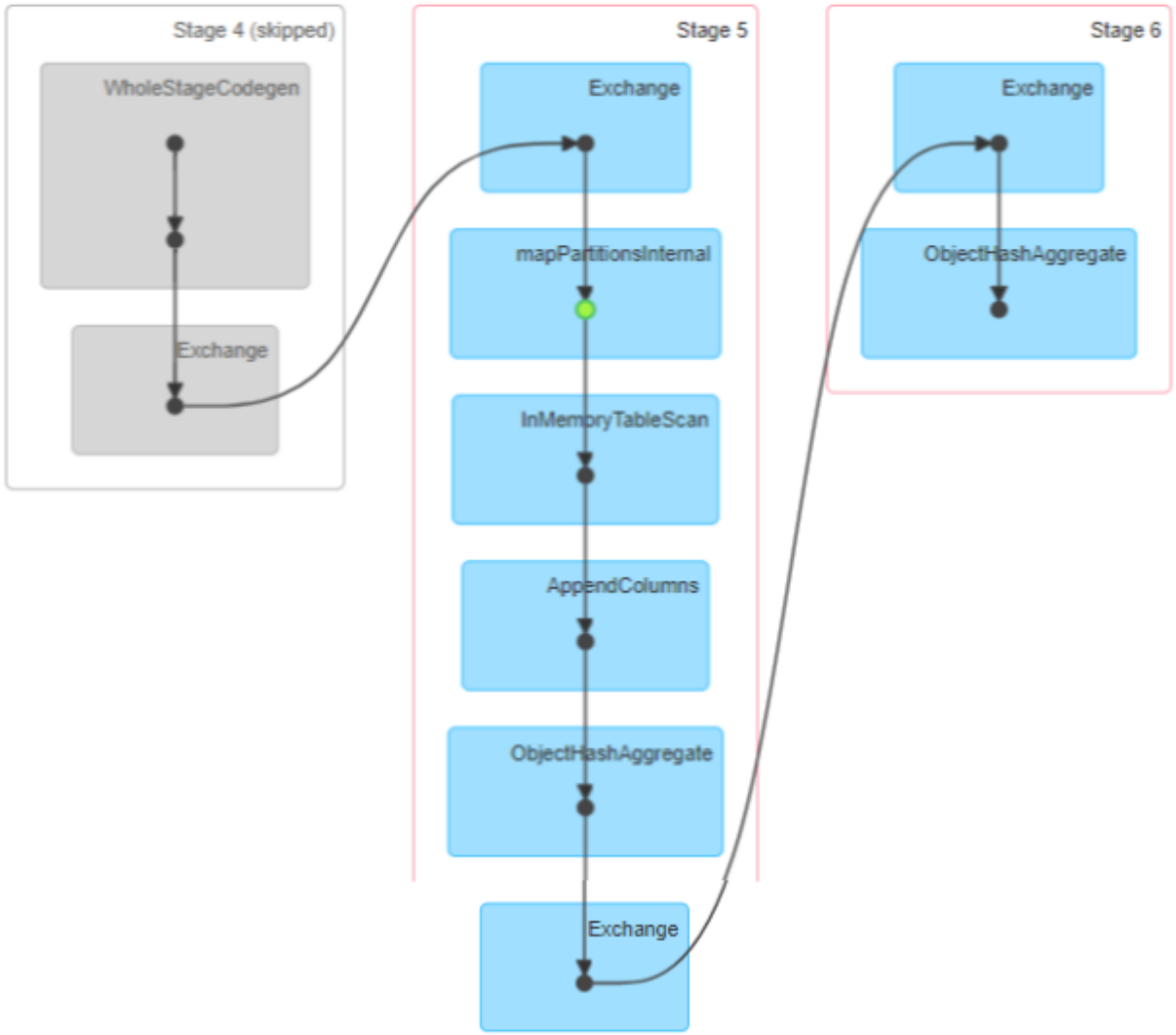


Below are the stage computation DAGs built by DAG scheduler for the two Jobs:





Stage Computation DAG for Job-1



Stage Computation DAG for Job-2

As clear from stages computation DAGs of the two Jobs, in stage computation DAG of Job-2, stage 4 is skipped for computation (Skipped stage is greyed out) since 'ShuffledRowRDD[11]' used in stage 5 of Job-2 is already

computed and cached in stage 2 of Job-1, Job-1 bring submitted earlier for execution.

Summary: It should be evident now, how the staging of execution flow in Spark provide modularization and resiliency to the overall execution of Spark application. Users can track stage wise progress of Spark application, can access several stage wise metrics to assess the stage execution efficiency. And finally, most importantly, the stage wise progress and associated metrics can lead to clues to Spark application optimization.

In case of further queries/doubts about Spark stages , or for any feedback for this story, please do write in the comments section.

Data Science | Programming | Technology | Software Development | Big Data