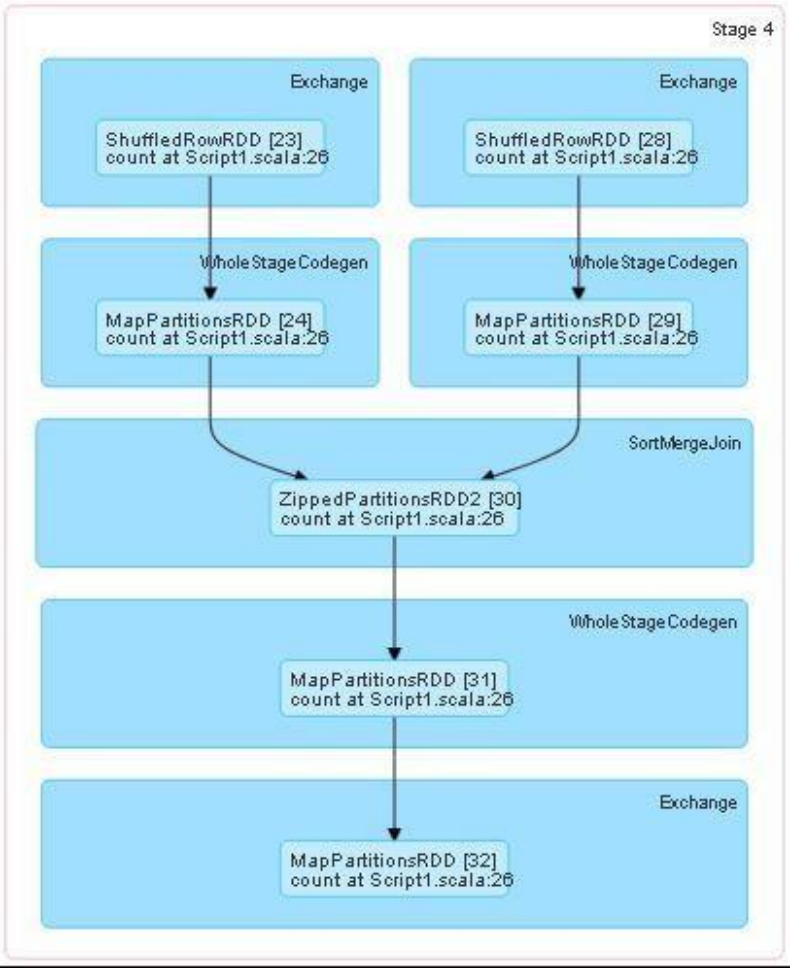
Script 1

In script1, data is read from two files student_data.csv and school_data.csv. These tables have a common attribute, i.e. school_id. Student_data table has 25,000 rows and school_data table has 10,000 rows. The scripts performs a full outer join on the two tables on the key 'school_id'. On the resulting table, distinct() function and count() function are applied to count the number of distinct records in the resulting table. Here, we note that a full outer join in Spark 2.4.5 is implemented as a SortMergeJoin.

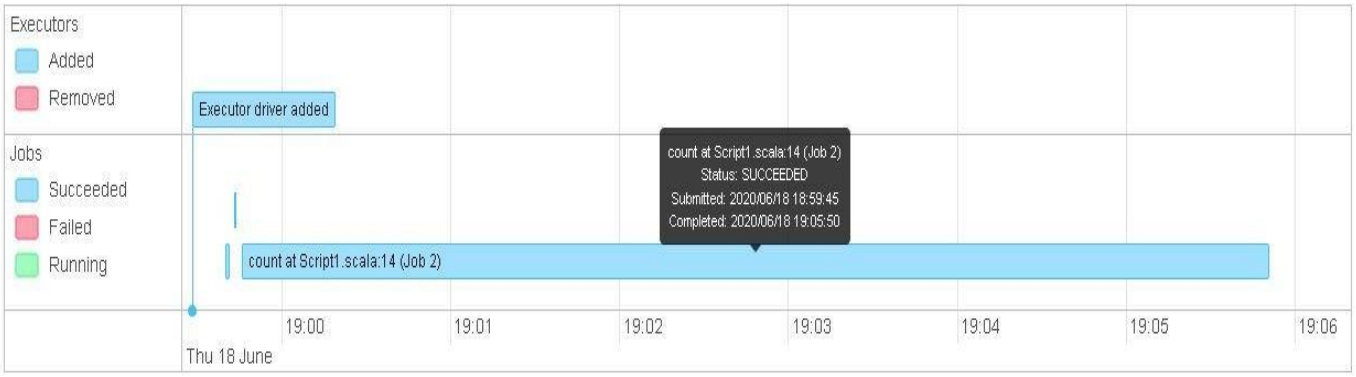


DAG of SortMergeJoin Stage

SortMergeJoin has 3 phases. In the Shuffle phase, the two tables are repartitioned as per the join keys. In the Sort phase, the data is sorted within each partition in parallel. Finally in the Merge phase, the two partitioned+sorted tables are merged. In the above DAG, you can see the SortMergeJoin happening as sort takes place in different partitions in parallel and joined again later in the stage.

Performance Tuning

We can improve the performance of this script1 using performance tuning strategies. In the original script1, we have setMaster() to “local” which means our script is running on only one core. In this case, our script takes a significant amount of time to run since there is no use of multithreading. The computation stage takes over 6 minutes which is quite high as shown in the Spark UI below.

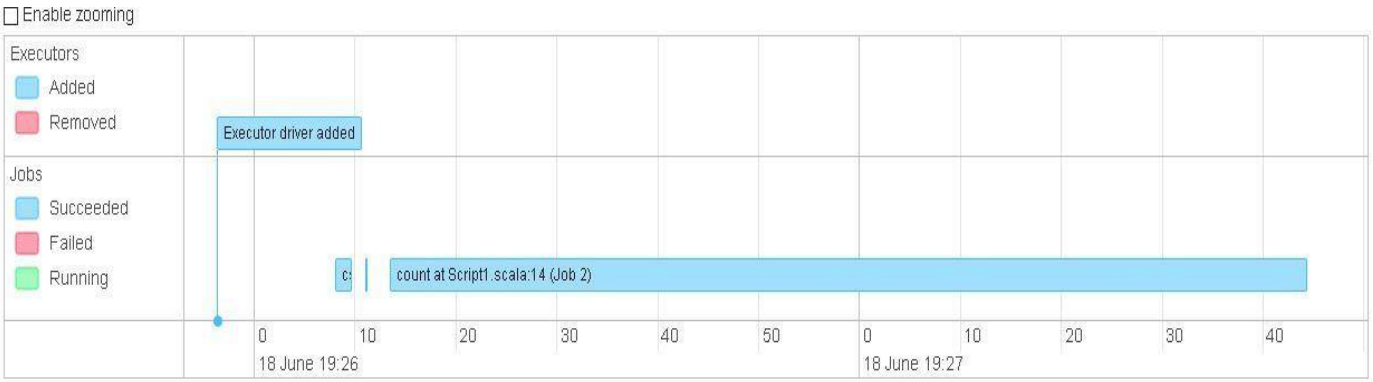


Completed Jobs (3)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at Script1.scala:14 count at Script1.scala:14	2020/06/18 18:59:45	6.1 min	5/5	403/403
1	csv at Script1.scala:12 csv at Script1.scala:12	2020/06/18 18:59:42	0.2 s	1/1	1/1
0	csv at Script1.scala:11 csv at Script1.scala:11	2020/06/18 18:59:39	2 s	1/1	1/1

Spark UI for Script1 before Tuning

Now we set setMaster() to “local[*]” and see a considerable improvement in performance. The * signifies the number of cores on the computer. We can set it to 3,4,etc as well. The time taken for the operations is now significant lesser at 1.5 minutes. Almost one fourth of the time taken earlier! This improvement in performance is due to multithreading that has been enabled in the second case.



Completed Jobs (3)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at Script1.scala:14 count at Script1.scala:14	2020/06/18 19:26:13	1.5 min	5/5	403/403
1	csv at Script1.scala:12 csv at Script1.scala:12	2020/06/18 19:26:10	0.1 s	1/1	1/1
0	csv at Script1.scala:11 csv at Script1.scala:11	2020/06/18 19:26:07	2 s	1/1	1/1

Spark UI after tuning

Thus, we conclude that multithreading when used correctly can lead to a significant improvement in runtime.