**Pipeline Shuffle: Reducing Shuffle I/O Latency in Spark**

Apache Spark runs job stage by stage. The stages are built up and separated by the *DAGScheduler* according to the RDD’s *ShuffleDependency*. Currently, the shuffle process is time-consuming due to the I/O cost for writing data to the storage. More precisely, the shuffle senders need to write the data to disk first, and then the data can be shuffled to the receivers.

The main idea of Pipeline Shuffle is to break down the barrier between Stages in Spark. The key insight is to pre-fetch data once produced by the senders. To achieve this, we need to do following modifications on existing Spark platform.

1. **Design and Implementation of Pipeline Shuffle**
2. **Generate the DAG**

When a job is submitted, we will find the final stage of the job, and allocate executor for each task randomly (which may not be a good idea). After that, we backtrack the final stage to its parent stages, and then grandparent stages, etc. recursively. If the dependency type is a *ShuffleDependency*, we will update the total map partition number (i.e. the partition number of RDD with ShuffleDependency), ShuffleId, reduce task locations to *ReduceStatus*.

After that, we register *ReduceStatus* information to the MapOutputTracker view two newly added function named *getParentStagesWithReduceRegister()* and *getParentStagesAndId-WithReduceRegister()* in DAGScheduler. Basically, we maintain a stack here to store the visited RDD information and the corresponding ReduceStatus() information of previous stages (i.e. parent stages) recursively. Therefore, the tasks of the parents of a RDD that depends on multiple ShuffleDependecywill run with the ReduceStatus that has the same reduce tasks locations.

1. **Schedule the stage**

In pipeline shuffle, we submit a stage when the missing parents are empty or when all its parents are running. Here we use a FIFO TaskSet Scheduler to ensure the task set of parents are scheduled before the children stage via *submitStage()* in *DAGScheduler*.

1. **Schedule the tasks**

For scheduling the tasks of a stage, we modify the *submitMissingTasks()* in the *DAGScheduler*.

If the current stage is a ShuffleMapStage, it means the corresponding ShuffleMapTask will start later. We fetch the *ReduceStatus* of *ShulffeId* and notify the corresponding *BlockManager* to get ready for storage communication via Akka. On the other hand, we add *ShuffleId* via *setShuffleId()* as a new parameter into the corresponding task in this ShuffleMapStage.

If the current stage is a ResultStage, we will figure out whether it has Shuffle dependency. If it has at least one shuffle dependency, we will ask *MapOutputTracker* to fetch the corresponding ReduceStatus. And we start the tasks of this ResultStage according to the location information in the *ReduceStatus*, or using the location set by the default Spark’s task allocation function.

1. **Storage Layer Communication**

There is some metadata communication in Storage-Layer of the Spark cluster. We list the following key notification signals in our PipelineShuffle project.

*registerShuffePipe():*

It is a function added in *BlockManagerMaster().* This function will be triggered if the ShuffleMapTask is scheduled in submitMissingTasks. We pass the *ShuffleID* and the array of *ReduceStatus* to the *BlockManagerMaster().* And then *BlockManagerMaster()* will merge the *ReduceStatus* by the executor. This merge process is to reduce the communication overhead of storage-level metadata over the network.

When the *BlockManager* of a slave received the *RegisterShufflePipe* information, this slave will fetch the *ReduceStatues* according to the received *ShuffleID* through the *getReduceStatuses()* function we build in *MapOutputTracker* module. Therefore, the slave will get the information of total map partition of this shuffle. And then the slave will generate the corresponding buffers to store the metadata of *ReduceStatues*. The array is HashMap indexed by ShuffleID. The value of the HashMap is an array with number of total-map-partition LinkedBlockingQueue to store the FetchResults.

*pipeEnd():*

This function will be triggered when a ShuffleMapTask is finished on the executor. The executor will call the local *BlockManager* to send the *pipeEnd* message to other slaves. Instead of block in current stage, we start a new thread on the executor to send out this *pipeEnd* message. The thread will first merge the *ReduceStatus* by the *executorId* to reduce the network traffic. And then the thread filters the local partition and calls the local *pipeEnd()* directly. After that, the thread will send RPC call to the remote *BlockManager* to notify the *pipeEnd* message.

When a remote *BlockManager* (i.e. reducer) receives the *pipeEnd* message, the reducer will find out the reduce partition on itself for this shuffle, and calculate the size of each reduce partition after this map stage. After that, the reducer will call *sendRequest()* to fetch the corresponding data block it needs.

*sendRequest():*

It is an asynchronous function for the reducer to fetch the data block. The blocks are divided by their locations and sizes. And here we build an array of *shuffleFetchResultQueue* (a.k.a. *LinkedBlockingQueue*) on each reduce executor for maintaining metadata of data blocks that need to be fetched.

If the needed data block is empty, this *sendRequest(*) function will just put a EmptyFetchResult to the corresponding *LinkedBlockingQueue*. If the block is non-empty and local, instead of fetch the data block immediately, we will put a LocalFetchResulton the *shuffleFetchResultQueue.* If the block is non-empty and remote, we will send a fetch request via *shuffleClient* to fetch the block, the FetchResult of the remote block will be added to the *shuffleFetchResultQueue* as soon as the remote data fetching is finished. When the remote fetch finished, the data will be cached in the local disk at the same time for future data reading via *putCachedBlock().*

1. **Shuffle read**

Shuffle Read is handled by the *BlockStoreShuffleReader* in Spark Shuffle module. At first, it will check whether the shuffle data is cached. If the data is cached locally, the *BlockStoreShuffleReader* will ask the local *BlockManager* to get the total map partition number and the array of *LinkedBlockingQueue*. Since the fetch process is block by block, we can ensure the “size” of each *LinkedBlockingQueue* is equals to the number of map partitions. In other words, the total number of request that the reader should wait is mapPartition \* (endPartition – startPartition). Then it uses these two parameters (i.e. endPartition & startPartition) to create a new *ShuffleBlockFetchIterator*. And then we filter the NULL input stream in *ShuffleBlockFetchIterator* to get the final key-value pair iterator.

Inside the *ShuffleBlockFetchIterator*, if it's a cached one (i.e. Those created with map size and the array of queue), it will just wait for the request to become available and return the (BlockId, InputStream) pair. If the fetch result is an EmptyFetchResult, it will just return a “null” as InputStream. If the result is a LocalFetchResult, it will ask the local *BlockManager* for the data and return. If the result is a SuceessFetchResult (i.e. the remote non-empty fetch), it will return the InputStream filled with buffer at once. This all happens when the data is consumed on calculation of the task.

1. **Performance Evaluation**

We run some basic benchmark tests of Pipeline Shuffle, compared with original Apache Spark. **(1) GroupByTest**

We conduct GroupByTest with clusters of 20 slave nodes and 50 slave nodes separately. We build the Spark cluster on AWS EC2 with instance type of m1.xlarge. Each instance node has 2 CPU core, 7.5 GB main memory and 420 GB storage.

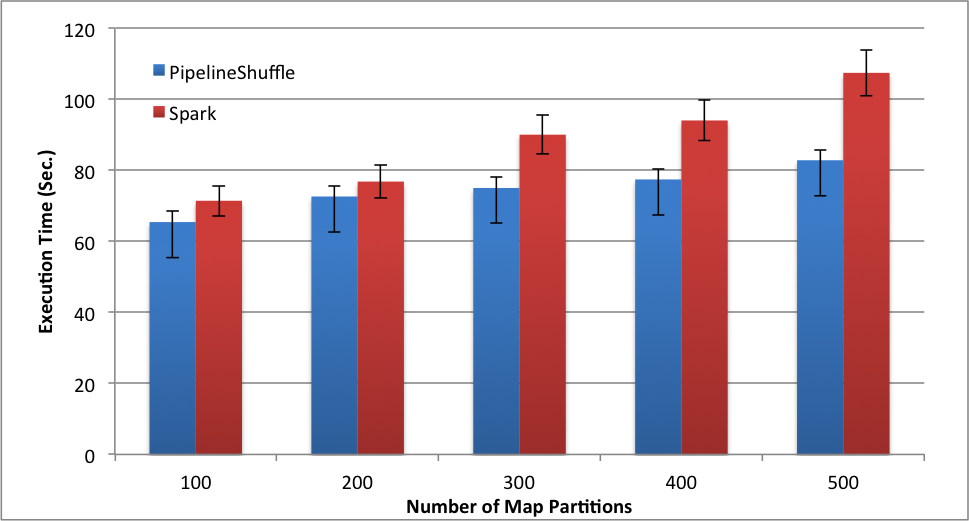


Figure 1. GroupByTest in a 20-slave cluster

As shown in Figure 1, we implemented Pipeline Shuffle on a 20-slave cluster with map partition number ranging from 100 to 500, 10000 key-value pair, KeySize of 1000, 80 reduce partitions. Therefore the data size is ranging from 100\*10000\*1000 = 1 GB to 500\*10000\*1000 = 5 GB. In Figure 1, PipelineShuffle can reduce 10% – 24% execution time of Spark. With larger dataset, the performance gain is more significant.

However, when we directly extend our PipelineShuffle into a 50 Slaves Cluster with 20 GB dataset (1000 Map Partitions, 10000 key-value pair, KeySize of 2000, 100 Reduce Partitions), the performance gain of PipelineShuffle decreases, sometimes is even negligible. We run the GroupByTest for 5 rounds, the result is depicted as Figure 2. As shown in Figure 2, in the test round 3 – 5, the PipelineShuffleTotal execution time is roughly equal to the original SparkTotal running time.

After we analyze the performance, we figure out the Reduce stage of our PipelineShuffle is the straggler. More precisely, the pre-scheduling on tasks of successive stages is time-consuming when the number of reduce partitions is large. The key problem is, before the offer was made for running tasks of next stage, the TaskScheduler needs to scan all the tasks on the waiting-list and evaluates the data locality (from process\_local, then node\_local, etc.). And this data locality scanning is high overhead when the number of reduce-partitions and number of nodes are large.

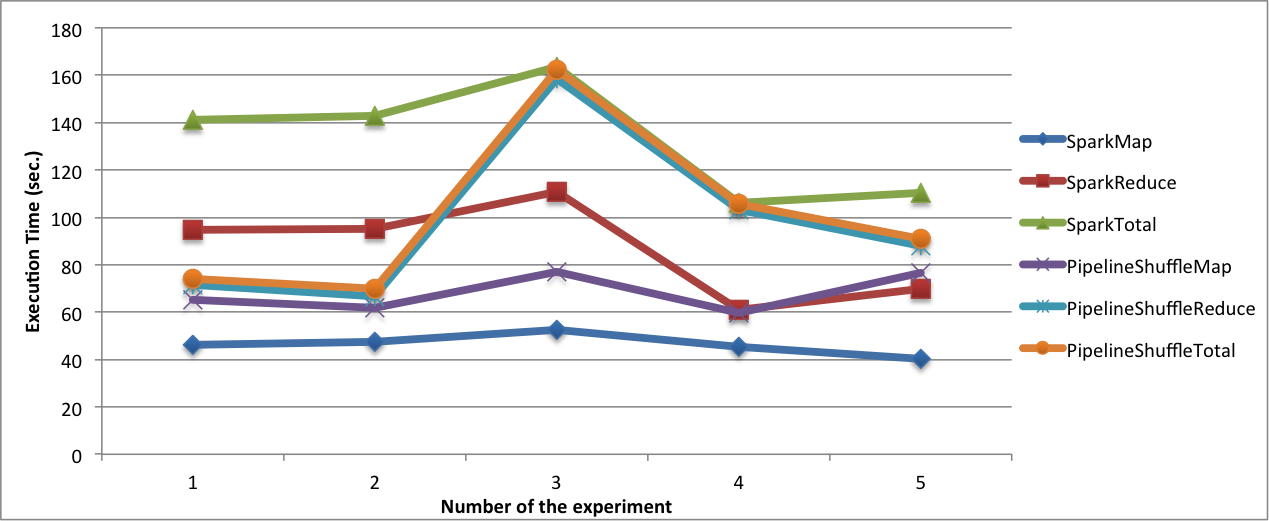


Figure 2. GroupByTest in a 50-Slave Cluster (20 GB Dataset)

To solve this problem, we disable the pre-scheduling of tasks in successive stages, and only enable data pre-fetching. As shown in Figure 3, with the modification of pre-fetch without pre-scheduling (i.e. *PipeShuffle Without Pre-Scheduling* in the figure), the reduce stage is 10x faster than original Spark. The total execution time of PipelineShuffle is less than HALF of original Spark.

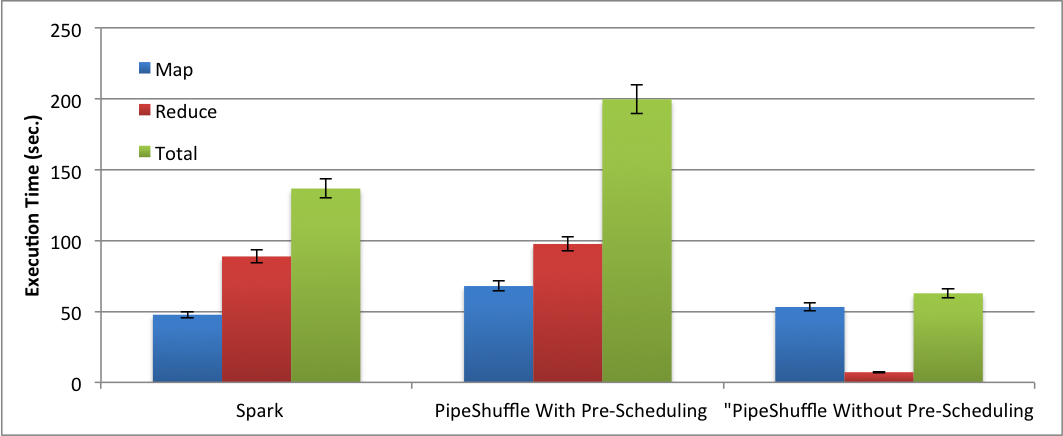


Figure 3. Performance Comparison of Spark, PipelineShuffle with/without Pre-scheduling

**(2) WordCount**

Next, we evaluate the PipelineShuffle’s performance on WordCount. We run wordcount of 6GB dataset with a 50-slave cluster. As shown in Figure 4, the PipelineShuffle do not have much performance gain compared with default Spark.

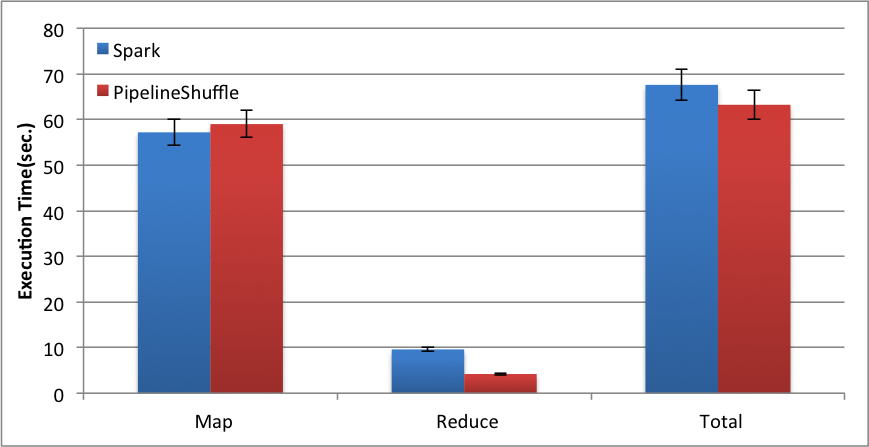


Figure 4. WordCount performance result of Spark and PipelineShuffle

PipelineShuffle can decrease the execution time of reduce process. However, in wordcount, the running time of reduce stage is relatively negligible. Even though reduce stage in PipelineShuffle is 2x faster than it in Spark. PipelineShuffle’s performance gain for total execution time is around only 7%.