

# Title

Zhouwang Fu<sup>1</sup> and Zhengwei Qi<sup>1</sup>

<sup>1</sup>Shanghai Jiao Tong University

## Abstract

Shuffle is the term used to describe the cross-network read and aggregation of partitioned ancestor data before invoking reduce operation. As DAG computing frameworks keep evolving, calculation and scheduling of each task are well optimized. However shuffle cuts off the data processing pipeline, introduce significant latency to successors. To remove shuffle overhead, we present XXX, a plugin system to decouple shuffle from DAG computing framework. XXX captures shuffle data in the memory and uses heuristic-FIFO scheduling to balance data blocks to eliminate the explicit barrier. We implement XXX and change Spark to use XXX as external shuffle service and scheduler. We evaluate XXX performance both on simulation and 50-machine Amazon EC2 cluster. Results show that, by incorporating XXX in Spark, the shuffle overhead can be reduced XXX.

## 1 Introduction

## 2 Motivation

In this section, we first study the shuffle pattern (2.1). Then we show the observations of the opportunities to optimize shuffle in 2.2

### 2.1 Characteristic of Shuffle

In large scale data parallel computing, enormous datasets are partitioned into pieces to fit the memory of each node since the very beginning of MapReduce[8]. Meanwhile, complicated application procedures are divided into steps. The successor steps take the output of ancestors as input to do the computation. Shuffle occurs when each successor needs part of data from all ancestors' output. In order to provide a clear illustration, we define those computing each partition of data in one step as task. For tasks that generate shuffle output, we call them map task. For tasks that consume shuffle output, we call them reduce tasks. Note that one task may have both shuffle data generation and consumption, we call it intermediate task.

Shuffle is designed to achieve an all-to-all data blocks transfer among nodes in cluster. It exists in both MapReduce models and DAG computation models.

Shuffle mainly contains two phases itself: **Data Partition** and **Data Transfer**. For **Data Partition**, each map task and intermediate task will partition the result data (key, value pair) into several buckets according to the partition function. The buckets number equals to the number of tasks in the next step. When the map tasks and intermediate tasks finish, all the shuffle output data will be written into local persistent storage for fault tolerance [8, 10]. **Data Transfer** starts at the beginning of reduce tasks and intermediate tasks. These tasks will fetch the data that belongs to their corresponding partitions from both remote nodes and local storage.

In short, shuffle is loosely coupled with application context and it's I/O intensive.

Since intensive I/O operation will be triggered during a shuffle, this can introduce a significant latency to the application. Reports show that, 60% of MapReduce jobs at Yahoo and 20% at Facebook are shuffle intensive workloads[4]. For those shuffle intensive jobs, the shuffle latency may even dominate Job Completion Time. For instance, a MapReduce trace analysis from Facebook shows that shuffle accounts for 33% JCT on average, up to 70% in shuffle intensive jobs[6]. Meanwhile, the completion time of shuffle correlates with the performance of storage devices, network and even applications. This variation may bring a huge challenge for operators to find the correct configuration of the DAG framework.

### 2.2 Observation

Of course, shuffle is unavoidable in a DAG computing process. But *can we mitigate or even remove the overhead of shuffle?* To find the answers, we run some representative applications on a Spark in a 5 m4.xlarge Amazon EC2 cluster. We then capture and plot the CPU utilization, I/O throughput and tasks execution information on each node. Take the trace in Figure 1 as an example, which is captured during one Spark GroupByTest job. This job has 2 rounds of tasks for each node. We mark the 'Execution' phase in the figure from the launch time of the first task on this node to the execution finish timestamp of the last



Figure 1: CPU utilization and I/O throughput of a node during a Spark single shuffle application

one. The 'Shuffle Write' phase is marked from the timestamp of the beginning of the first partitioned data write. The 'Shuffle Read and Execution' phase is marked from the start of the first reduce launch timestamp. Figure 1 contains data including two stages connected by one shuffle. By analyzing the trace combining with Spark, we propose following observations.

### 2.2.1 Multi-rounds tasks in each stages

Both experience and DAG framework manuals recommend that multi-rounds execution of each stage will benefit the performance of whole application. For example, Hadoop MapReduce Tutorial [2] suggests that *10-100 maps per node* and *0.95 or  $1.75 \times \text{no. of nodes} \times \text{no. of maximum container per node}$*  seem to be the right level of parallelism. In Spark, the memory available to each task (partition) roughly equals to  $\text{spark.executor.memory} \times \text{spark.memory.fraction}$  [3]. We have two rounds of tasks in job of Figure 1 to process about 70GB data. Figure 1 shows that the second phase of shuffle – **Data Transfer** will start until the reduce stage starts. But the shuffle data will become available as soon as the execution of one task is finished. Though in the context of Spark, the reduce task can do computation while fetching data, the uncontrolled network congestion may still hurt the performance. However, if the destination of the shuffle output of each task is aware, the property of multi-round can be leveraged to do **Data Transfer** ahead of reduce stage.

### 2.2.2 Tight couple between shuffle and computation

Another information we get from the trace is that shuffle should be decoupled from task which is an execution unit in both Spark and Hadoop MapReduce. In general, CPU and memory are bound as a schedule slot in DAG resource scheduler. When a task is scheduled to a slot, it won't release until it reaches the end of task. In Figure 1, the resource of Spark executor will be released at the ending of 'Shuffle Write'. But CPU becomes idle almost

as soon as the 'Execution' is finished. On the other hand, shuffle is I/O intensive job. It doesn't involve CPU and application context. If the shuffle can be decoupled from task, the slot can be released after 'Execution' phase. The early release can benefit other tasks to achieve better overall performance of the DAG framework.

### 2.2.3 I/O performance varies

When we look into the performance of disk and network in our test case, there is huge variance. Since we use the standard EBS as our backend storage for the EC2 instances, the I/O performance of disk is poor. At the same time, the exclusive bandwidth of each instance is 750 Mbps [1]. In this case, the bottleneck of shuffle is disk, which introduces a significant latency for the application. Vice versa, in some cases, the congestion of network may also become the bottleneck of shuffle [7]. The uncertainty of the I/O performance causes a huge challenge for optimizing the DAG computing in the cluster. For network latency, the most we can do is to mitigate the transfer delay. As for disk write, we believe it's not necessary for today's cluster. Recall that the persistence of shuffle data is used only for reduce fault tolerance, but mean time to failure (MTTF) for a server is counted in the scale of years [9]. In addition, we believe combining the high speed of network and memory is a better choice for fault tolerance. We will present more details in Section 4.

### 2.2.4 Shuffle size is small

Comparing to the whole input dataset, the size of shuffle data is relatively small. For instance, in a 10GB input TeraSort, the shuffle size is only 4GB. When it's mapped to a 5 nodes cluster, it only takes about 800MB memory to cache the shuffle data in memory. This is another reason that disk should not be involved in the whole shuffle procedure.

Based on these observations, it's straightforward to come up with an optimization to use memory to store the

shuffle data and overlap the I/O operation of shuffle by leveraging multi-rounds property of DAG computing. In order to achieve this optimization, we have to decouple shuffle from task and perform pre-fetch as soon as each output of map task and intermediate task is available. But is this feasible? We try to answer this question in the following sections.

## 3 Achieve Shuffle Pre-fetch

### 3.1 Decouple shuffle from task

Instead of writing shuffle data into disk, the data can be intercepted in memory and directly written to network. But where?

### 3.2 Pre-schedule with Application Context

Naively random map tasks to end hosts. It's bad, may skew. We have perform more balanced allocation to avoid hurting the performance of successor tasks. Show the simulation with open cloud trace.

In order to achieve better balanced allocation, we should combine with application context and DAG.

For one shuffle, we can use first few tasks output to predict reduce distribution such as [5].

But results vary due to the input data distribution and partition function. Show three pics (hash partition and two range partition)

For hash partition function, in most scenarios are enough to have first completed tasks.

For range partition function and customized partition function, the relation between one output and the whole distribution can be opposite. To avoid complex modification, we keep the partition function as a black box and use weighted reservoir sampling to probe the distribution. Show the prediction result and accuracy.

For each prediction result, a percentage array of total data composition is calculated.

Combined with DAG information, i.e. other co-existing shuffle dependencies.

present pseudo code. (not completed yet).

## 4 Design

## 5 Evaluation

## 6 Conclusion

## References

- [1] Amazon ec2. <https://aws.amazon.com/ec2/>.
- [2] Apache hadoop tutorial. <http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html>.
- [3] Apache spark 1.6.2 configuration. <http://spark.apache.org/docs/1.6.2/configuration.html>.
- [4] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *USENIX Annual Technical Conference*, pages 1–12, 2014.
- [5] D. Cheng, J. Rao, Y. Guo, and X. Zhou. Improving mapreduce performance in heterogeneous environments with adaptive task tuning. In *Proceedings of the 15th International Middleware Conference*, pages 97–108. ACM, 2014.
- [6] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 98–109. ACM, 2011.
- [7] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varies. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 443–454. ACM, 2014.
- [8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [9] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM, 2014.
- [10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant

abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.