Title

Zhouwang Fu[1] and Zhengwei Qi[1]
[1]*Shanghai Jiao Tong University*

Abstract

Shuffle is the term used to descirbe the cross-network read and aggregation of partitioned ancestor data before invoking reduce operation. As DAG computing framworks 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 elimite the explict 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 reduce 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 beginnig of MapReduce[11]. 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 generates shuffle output, we call them map task. For tasks that consume shuffle output, we call them reudce 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 [11, 19]. **Data Transfer** can be further divied into two parts: **Shuffle Write** and **Shuffle Read**. **Shuffle Write** starts after execution of map tasks and intermediate tasks. Partitioned data will be stored on disk during **Shuffle Write**. **Shuffle Read** 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 lantency to the application. Reports show that, 60% of MapReduce jobs at Yahoo and 20% at Facebook are shuffle intensive workloads[6]. For those shuffle intensive jobs, the shuffle lantency 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[9]. 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 Aamzon EC2 cluster. We than 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 capu-

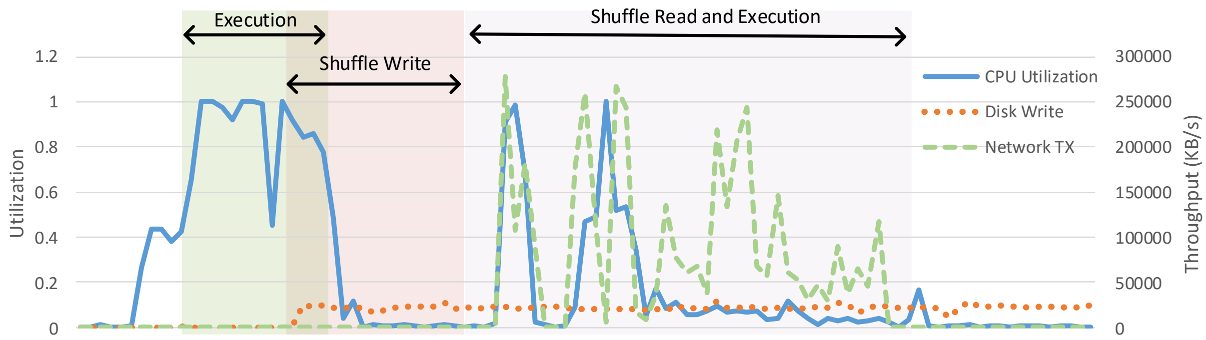


Figure 1: CPU utiliazation and I/O throughput of a node during a Spark single shuffle application

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

2.2.1 Multi-rounds tasks in each stages

Both experiece and DAG framework manuals recommand that multi-rounds execution of each stage will benifit the performance of whole application. For example, Hadoop MapReduce Tutorial [2] suggests that *10-100 maps per-node* and *0.95 or 1.75* $\times$ *no. of nodes* $\times$ *no. of maximum container per node* seem to be the right level of parallelism. Spark Configuration also recommends 2-3 tasks per CPU core in the cluster[4]. 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 compuation 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 mutil-round can be leveraged to do **Data Transfer** ahead of reduce stage.

2.2.2 Tight copule between shuffle and computation

Another information we get from the trace is that shffle should be decoupled from task which is a execution unit in both Spark and Hadoop MapReduce. In general, CPU and memory are binded 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 realsed at the ending of 'Shffle Write'. But CPU becomes idle almost as soon as the 'Exectuion' is finished. On the other hand, shuffle is I/O intensive job. It doesn't involved CPU and application context. If the shffle can be decopled from task, the slot can be relased after 'Execution' phase. The early release can benifit other tasks to achieve better overall performance of the DAG framework.

2.2.3 I/O performance varys

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 bolltle of shuffle[10]. The uncertainty of the I/O performance cause a huge challenge for optmizing 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 tolerence, but mean time to failure(MTTF) for a server is counted in the scale of year[14]. In addition, we believe combing the high speed of network and memory is a better choice for fault tolerence. We will present more details in Section 4.

2.2.4 Shuffle size is small

In order to accelearte computation, Spark will put all the input data set for a task into memroy. Comparing to the input dataset, size of shuffle data is relatively small. We persent to typical application on Spark to show the releationship between shuffle data comparing with the input dataset in Figure 2. Although TeraSort[15] is known as a shuffle intensive job, in a 10GB input TeraSort, the shuffle size is less than 3GB. When it's mapped to a 5 nodes cluster, it only taks about 500MB memory (25% of input size for each node) to cache the shuffle data in memory. The data reported in [16] also shows that the amount of data shuffled is less than input data, by as much as a factor of 5-10. 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 a optimization to use memory to store the shuffle data and overlap the I/O operations 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 Optimization

In this section, we try to achieve shuffle optimization by applying

- Decouple shuffle from task
- Pre-fetch shuffle to reduce node

on the DAG computing framwork. We choose Spark as the representative of DAG computing framwork to implement our optimization.

3.1 Decouple shuffle from task

On the map task side of shuffle, it's used to partition the output of map task according to the pre-defined partitioner. More specifically, shuffle takes a set of key-value pairs as input. And than it calculates the partitioner number of a key-value pait by applying pre-defined the partition function to the key. At last it put the key-value pair into the coressponding parition. The output at last is a set of blocks. Each of them contains the key-value pairs for one partition. For those application context unrelated blocks, they can be easily hijacked in the memory of Spark executor and moved out of JVM space via memory mapping. Meanwhile, we have to prevent the memory spill during the shuffle partition procedure, so that the shuffle data can never touch the disk. The default shuffle spill threshold in Spark is 5GB[3], which is big enough in most scenarios according to Section 2.2.4.

3.2 Pre-schedule with Application Context

When the shuffle output blocks are available in memory, they can be pre-fetched to the remote hosts to hide the network transfer time. But at that time, the reduce tasks taking shuffle as input are still pending. In other word, the remote hosts of those blocks keep unknown until the reduce tasks are scheduled by the DAG framework. In order to break this serialization between map tasks and shuffle, we have to first pre-schedule the task-node mapping ahead of DAG framework scheduler. We explore several pre-scheduling schemes in different scenrios. And evalute the performance of pre-scheduling and prediction by calculating the improvment of reduce tasks completion time with trace of OpenCloud[5]. We first emulate the sheduling algorithm of Spark to schedule the reduce tasks of one job, and take the bottleneck of the task set as the completion time. Then we remove the shuffle read time as the assumption of shffle data pre-fetch and emulate under different schemes. The result is shown in 3b. Note that since most of the traces from OpenCloud is shuffle-light workload as showen in Figure 3a. The average shuffle read time is 0.6% of total reduce completion time. So we will only use this trace to evalute the pre-scheduling and prediction schemes.

3.2.1 Random Task-Node Mapping

The simplest way of pre-scheduling is mapping tasks to different nodes evenly. As shown in Figure 3b, Random mapping works well when there is only one round of tasks in cluster. But multi-round in cluster is overwhleming according to Section 2.2.1. The performance of random mapping collapses as the round number grows. After analyzing the trace, we find out that it's caused by data-skew. Reports in these papers[13, 7, 12] also claim that data-skew is commonly exist in data-parallel computing. When we apply a random mapping, it's probable to assign sevearl slow tasks on one node. The collision than slow down the the whole stage, which make the performance even worse than those without shuffle-prefetch.

3.2.2 Schedule with Heuristic

The failure of random mapping was obvious caused by application context (e.g. Shuffle input of each task) unawareness, which results in a heaver data skew. To avoid the 'bad' schedule results, we have to levrage the application context as assistance. The optimal schedule decision can be made under the awareness of shuffle dependecy number with input size for each task. But these data is unavialable when the pre-fetch starts. It's impossible to reach the optimal.

According to the DAG computing process, the shuffle size of each reduce task is decided by input data map task computation and hash partitioner. And for each map task, it will produce a data block for each reduce tasks, like '1-1' in Figure 4. '1-1' means it's produced by 'Map Task 1' for 'Reduce Task 1'. For Hadoop MapReduce, the shuffle

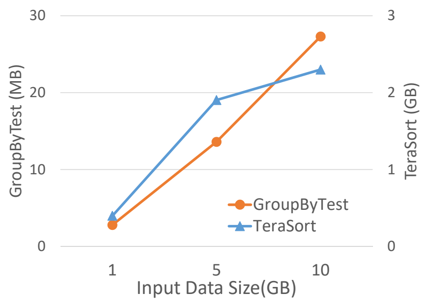
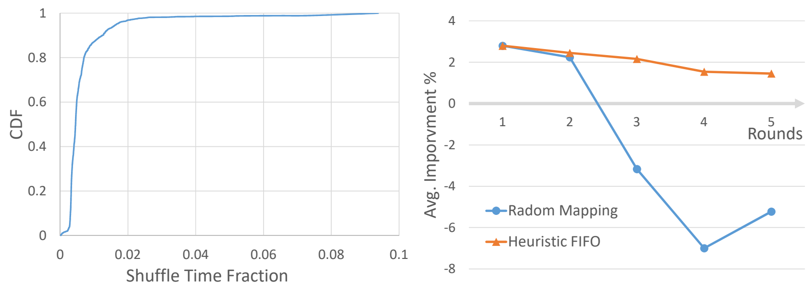


Figure 2: Shuffle Size Comparing with Input Size



(a) Shuffle Time Fraction CDF

(b) Stage Completion Time Improvement

Figure 3: Emulate Result of OpenCloud Trace

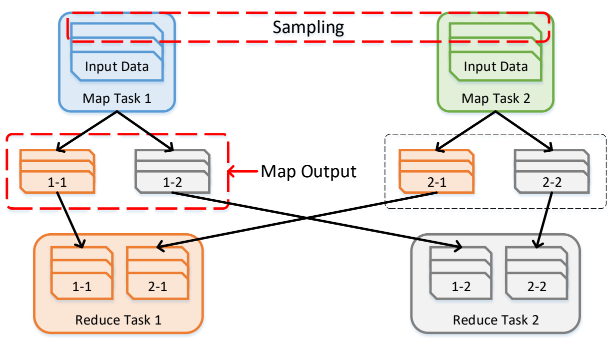


Figure 4: Shuffle Data Prediction

input for each reduce task can be predicted with decent accuracy[8]. They propose a liner regression model based on Verma et al.[17] that the ratio of map output size and input size is invariant given the same job configuration. Several map outputs (marked as Map Output in Figure 4) are picked as observation objects to train the model and than predict the final reduce distribution. But in the more sophisticated DAG computing framwork like Spark, this model can't fit. For instance, the reduce stage in Spark has more number of tasks that consume shuffle data instead of Hadoop MapReduce. More importantly, the customed partitioner can bring huge inconsistency between observed map output blocks distribution and the final reduce distribution, as we presented in Figure 5. We use different datasets with different partitioners to find the connection among three factors. We normalize threes sets of data to [0,1] to fit in one figure. In Figure 5a, we use a random input dataset with the Hash Partitioner of Spark[3]. In Figure 5b and Figure 5c, we use a skew dataset with the Range Partitioner of Spark[3]. We randomly pick one observation map output and plot. As we can see, in hash partitioner, the distribution of each map(blue area) is close to the final reduce distribution(orange boxes). The prediction results also turns out well fitted. As we apply linear regression model to predict the final reduce distribuiton of Range Partitioner. The prediction is severely effected by the skew observed map output distribution.

To avoid this inconsistency in some cases, we introduce another methodology, weighted reservoir sampling, to mitigate this inconsistency. The classic reservoir sampling is designed for randomly chossing k samples from n items, where n is either a very large or unknown number[18]. For each partition of data that produce shuffle output, we use reservoir sampling to randomly pick $s \times p$ of samples, where p is the partition number of input data and s is a tunable number. In Figure 5c, we set $s = 3$. Afther that, the map function is called locally to process the sampled data. As the 'Sampling' part shown in Figure 4, the final sampling map outputs are collected with the size of each parition of input data which is used as weight for each set of sample. For each reduce, the predicted size

$$size_i = \sum_{j=0}^{m} partitionSize_j \times \frac{sample_i}{s \times p}$$

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

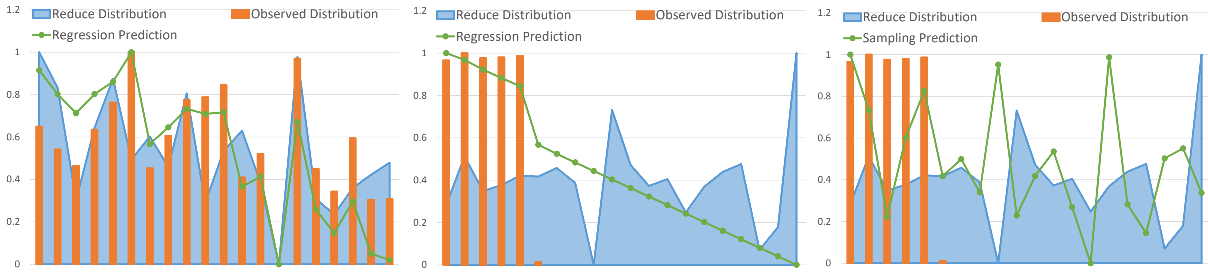
For hash partition function, in most scenrios are enough to have first completed tasks.

For range partition function and customed partition function, the relation between one output and the whole distribution can be oppsite. To avoid complex modification, we keep the partition function as a black box and use weighted reservior sampling to prob the distribution. Show the prediction result and accurance.

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).



(a) Linear Regression Prediction of Hash Partitioner

(b) Linear Regression Prediction of Range Partitioner

(c) Sampling Prediction of Range Partitioner

Figure 5: Reduction Distribution Prediction

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 source. `https://github.com/apache/spark/tree/branch-1.6`.

[4] Apache spark 1.6.2 configuration. `http://spark.apache.org/docs/1.6.2/configuration.html`.

[5] Opencloud hadoop cluster trace. `http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html`.

[6] 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.

[7] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, volume 10, page 24, 2010.

[8] 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.

[9] 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.

[10] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 443–454. ACM, 2014.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 522–533. IEEE, 2012.

[13] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.

[14] 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.

[15] O. OMalley. Terabyte sort on apache hadoop. *Yahoo, available online at: http://sortbenchmark.org/Yahoo-Hadoop. pdf,(May)*, pages 1–3, 2008.

[16] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI. Making sense of performance in data analytics frameworks. In *NSDI*, volume 15, pages 293–307, 2015.

[17] A. Verma, L. Cherkasova, and R. H. Campbell. Resource provisioning framework for mapreduce jobs with performance goals. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 165–186. Springer, 2011.

[18] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[19] 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.